UNIVERSITÀ DI TRENTO

Department of Information Engineering and Computer Science

Bachelor's Degree in Computer Science

Final Dissertation

Structural Variants Detection in Multiple Myeloma

Supervisor

Prof. Emilio Cusanelli

Student

Alessandro Gostoli

Academic year 2022/2023

Acknowledgements

Thanks to my supervisor Professor Emilio Cusanelli, who helped me throughout the development of this elaborate. Thanks to my tutor Tomàs, that guided me through this project. Thanks to my professor Jaime that introduced me to bioinformatic. Thanks to all my collegues that helped me in these four months, especially Dani. Im grateful to all the people mentioned, because they always supported me and went along with me in this journey. In the end, thanks to my family, because i can always count on them; they are always by my side and will never leave me.

Contents

Acronyms **3**

Abstract **4**

1 Introduction **5**

2 Literature review **7**
2.1 Structural Variants Calling . . . 7
2.1.1 Hystory . . . 7
2.1.2 Importance of Structural Variants . . . 7
2.1.3 Type of Structural Variant . . . 7
2.1.4 Detection of Structural Variants . . . 8
2.1.5 Structural Variants Pipeline . . . 9
2.2 Data . . . 9

3 Methods **11**
3.1 Tools . . . 11
3.1.1 Cluster . . . 11
3.1.2 Snakemake . . . 11
3.1.3 Conda . . . 11
3.1.4 Guppy . . . 11
3.1.5 Nanoplot . . . 11
3.1.6 Minimap2 . . . 11
3.1.7 Samtools . . . 12
3.1.8 Sniffles2 . . . 12
3.1.9 CuteSV . . . 12
3.1.10 BCFtools . . . 12
3.2 Pipeline . . . 12
3.2.1 Introduction . . . 12
3.2.2 Data . . . 12
3.2.3 Rules parameters . . . 13
3.2.4 Configuration . . . 13
3.2.5 Rule all . . . 14
3.2.6 Guppy . . . 14
3.2.7 Nanoplot . . . 15
3.2.8 Concat Guppy . . . 19
3.2.9 Concat Samples . . . 20
3.2.10 Minimap2 . . . 20
3.2.11 Samtools . . . 21
3.2.12 Sniffles2 . . . 23
3.2.13 CuteSV . . . 23
3.2.14 BCFtools . . . 24
3.2.15 Snakemake . . . 24

4 Results **25**

5 Discussion **26**

6 Conclusions **27**

Bibliography **27**

Acronyms

SV Structural Variants / Structural Variations

MM Multiple Myeloma

NGS Next-Generation Sequencing

CNV Copy Number Variants

MEI Mobile Element Insertions

ONT Oxford Nanopore Technologies

SNP Single Nucleotide Polymorphism

VCF Variant Call Format

SAM Sequence Alignment/Map

BAM Binary Alignment/Map

WGS Whole Genome Sequencing

WES Whole Exome Sequencing

Abstract

This thesis focuses on variant calling in cancer research, specifically the analysis of sequencing data and the detection of genetic variants in Multiple Myeloma. Utilizing a variety of bioinformatics tools and methodologies, the study explores the process of extracting genetic information from sequencing files and generating variant call files. The performance of different variant calling tools is evaluated, considering their sensitivity, specificity, and computational efficiency. Various filtering and post-processing strategies are examined to enhance the accuracy and reliability of variant calls. This research contributes to our understanding of genetic variations in cancer and their potential implications for diagnosis, prognosis, and targeted treatment approaches. I conducted this research project at the National Cancer Research Center of Madrid (CNIO) with the goal of detecting structural variations in the DNA of patients affected by Multiple Myeloma. To achieve this, I developed a pipeline using Snakemake, which is a sequence of software tools that guided me from the raw data, the whole genome sequence, to a VCF (Variant Call Format) file that contains information about the identified structural variations. The initial data for my project consisted of FAST5 files obtained from Altum Sequencing, which performed whole-genome sequencing on a multiple myeloma patient. Two separate sequencing runs were conducted using different flow cells. To begin the analysis, I performed base calling on the FAST5 files using guppybasecaller. Subsequently, I utilized Nanoplot to assess the quality of the base-called data. The next step in the pipeline involved concatenating the FASTQ files from each sequencing run into a single large FASTQ file. I then employed Minimap2 to align the reads from the combined FASTQ file to the reference genome, generating a SAM (Sequence Alignment/Map) file. The SAM file was subsequently converted to a BAM (Binary Alignment/Map) file, which was sorted and indexed for further analysis. With all the necessary files prepared, I employed two structural variation callers, Sniffles2 and CuteSV, to detect and characterize structural variations in the genome. These tools analyze the alignment information from the BAM file to identify structural variations such as insertions, deletions, duplications, and inversions. As a result of running the SV callers, I obtained multiple VCF files, each containing the identified structural variations for a particular caller. To extract relevant information from these files, I employed BCFtools, which allowed me to filter and explore the data. To facilitate the analysis of multiple VCF files obtained from different callers, I created a bash script that executed various BCFtools commands. This script streamlined the process and enabled me to repeat the analysis easily. The script collected all the relevant data in an Analyses.tsv file, where the caller names were set as column headers, and the extracted information was organized in rows. This format facilitated comparison and analysis of the results obtained from different callers. By following this pipeline, I was able to systematically detect and analyze structural variations in the DNA of Multiple Myeloma patients.

1 Introduction

Multiple myeloma is a malignancy of plasma cells that represents about 2% of all new cancer cases in the United Kingdom. The disease can cause the failure of the bone marrow leading to anaemia, immune paresis with resultant infection, bone pain and fractures, high calcium levels, and renal failure. In recent years, the therapeutic options to treat MM have rapidly expanded leading to increased length of survival. However, despite these improvements, MM remains an incurable condition for the majority of patients; most patients will die of their disease as it becomes refractory to treatment. Treatment can often help to control the condition for several years, but most cases of multiple myeloma can't be cured. Research is ongoing to try to find new treatments. [Bird and Boyd, 2019]

A paper in 2015 [Rustad et al., 2020], reported the first comprehensive study of SVs in Multiple Myeloma by WGS of sequential samples from 30 patients. Despite the limited sample set and the absence of gene expression data, the authors' findings indicated that SVs are a key missing piece to understand the driver landscape of multiple myeloma. Of particular interest, the authors of the study found a high prevalence of three main classes of complex SVs: chromothripsis, templated insertions, and chromoplexy. Overall, Rustad's study reveals the critical role of SVs in multiple myeloma pathogenesis.

Structural variation is generally defined as a region of DNA approximately 1 kb and larger in size and can include inversions and balanced translocations or genomic imbalances (insertions and deletions), commonly referred to as Copy Number Variants. These CNVs often overlap with segmental duplications, regions of DNA >1 kb present more than once in the genome, copies of which are > 90% identical. If present at >1% in a population a CNV may be called copy number polymorphism. Whole-Genome Sequencing studies have demonstrated the importance of Structural Variants in the initiation and progression of many cancers. [Nakagawa and Fujita, 2018] [Hamdan and Ewing, 2022] Functional implications of SVs include gene dosage effects from gain or loss of chromosomal material, gene regulatory effects such as super-enhancer hijacking, and gene fusions. The basic unit of SVs are pairs of breakpoints, classified as either deletion, tandem duplication, translocation, or inversion, which can manifest as simple events or form complex patterns where multiple SVs are acquired together, often involving multiple chromosomes. [Escaramís et al., 2015] The utilization of advanced Next-Generation Sequencing technology in clinical settings has provided a more comprehensive and detailed analysis of the genetics of Multiple Myeloma.

The NGS approach enables a more thorough classification of the disease's risk by identifying additional abnormalities that may have therapeutic implications. The evaluation of minimal residual disease (MRD) using NGS, which involves detecting mutated IGH and IGK gene rearrangements, has been incorporated into the Response Criteria by the International Myeloma Working Group and integrated into the guidelines of the National Comprehensive Cancer Network (NCCN).

Furthermore, both the European Medicines Agency (EMA) and the American Federal Drug Administration (FDA) have recommended the inclusion of MRD monitoring as a clinical endpoint in clinical trials. They also suggest the incorporation of MRD assessment in both ongoing and future trials. [Alt,]

Next-generation sequencing, also known as massively parallel or deep sequencing, refers to a DNA sequencing technology that has significantly transformed genomic research. This innovative approach allows for the rapid sequencing of an entire human genome within a single day. In contrast, the previous method of DNA sequencing, known as Sanger sequencing, required more than ten years to complete the final draft of the human genome. [Behjati and Tarpey, 2013]

Oxford Nanopore Technologies employs a long-read sequencing strategy based on the following principle: a single strand of DNA passes through a nanopore in a membrane while an electric voltage is applied across the membrane. As nucleotides traverse the pore, they impact the electrical resistance, enabling the measurement of current over time. This current signal, often referred to as the 'squiggle,' represents the raw data obtained by an ONT sequencer. Basecalling is the process of translating this raw signal into a corresponding DNA sequence. [Wick et al., 2019]

Long-read sequencing has advantages in detecting Structural Variants as it can span repetitive or challenging genomic regions. These longer reads, typically 5,000 base pairs or more, offer potential improvements in mapping accuracy and better capture of larger SVs compared to relying solely on short reads. However, both PacBio and Oxford Nanopore technologies, which generate long reads, have two main drawbacks. Firstly, sequencing costs are higher to achieve the same coverage as short-read sequencing. Secondly, the high error rate in long-read sequencing (around 8-20%) must be considered during alignment and SV calling processes. To tackle these challenges, specialized alignment methods such as BLASR, Minimap2, and NGMLR have been recently developed for long reads. In terms of SV identification, the field is still in its early stages, with only a few available methods. When working with long reads, SV detection methods are often tailored to specific sequencing technologies, primarily PacBio or Oxford Nanopore. [Mahmoud et al., 2019]

DNA sequencing is a vital process in genomics that allows researchers to determine the precise order of nucleotides in a DNA molecule. It has played a crucial role in understanding genetic information and unraveling the complexities of various biological processes. In the context of my project on detecting structural variations in Multiple Myeloma patients, DNA sequencing served as the foundation for analyzing the patient's genome.

The first step in the sequencing process is base calling. Base calling is the computational step that translates the raw electrical signal obtained from the nanopore sequencing technology, such as that used by Altum Sequencing, into a DNA sequence composed of nucleotides (A, C, G, T). This process is essential as it converts the electrical signals generated as the DNA passes through the nanopore into meaningful genetic information.

After base calling, the resulting data are released in the form of FASTQ files, which contain the DNA sequence reads along with their corresponding quality scores. To align these reads to the reference genome and determine their positions, a read mapping step was performed. In my pipeline, I used Minimap2, a versatile and efficient aligner, to compare the reads against the reference genome. This process allowed me to identify the locations where the reads originated from in the genome.

Once the reads were aligned, the pipeline moved on to the detection of structural variations. Structural variations are alterations in the DNA sequence that involve larger genomic rearrangements, such as deletions, insertions, duplications, and inversions. These variations can have significant implications in disease development and progression, including Multiple Myeloma.

To identify structural variations, I employed two specific structural variant callers: Sniffles2 and CuteSV. These tools analyze the aligned reads, taking into account discordant read pairs, split reads, and other supporting evidence, to detect and classify structural variations in the genome. By leveraging the alignment information from the BAM files, these callers could identify and characterize various types of structural variations in the DNA.

The output of the structural variant callers was in the form of VCF files, which store information about the detected structural variations, including their positions, sizes, and other relevant attributes. These VCF files served as the basis for further analysis and interpretation of the structural variations in the context of multiple myeloma.

Overall, my project utilized DNA sequencing, base calling, read mapping, and structural variant calling to explore the genomic landscape of multiple myeloma patients. These steps collectively provided insights into the structural variations present in the patients' DNA and contributed to a better understanding of the molecular mechanisms underlying the disease.

2 Literature review

2.1 Structural Variants Calling

2.1.1 Hystory

Until the middle of the past decade, it was thought that most of the common human genetic variation was due to single nucleotide polymorphisms (SNPs) and small insertion/deletion polymorphisms (indels), repetitive elements such as mini- and microsatellites, and small insertions or deletions that were detectable by traditional molecular biology techniques, while larger Structural Variants represented only rare events. [Escaramís et al., 2015]

About 15 years ago, thanks to the development of sequencing technologies it was finally possible to perform finer analysis of genome SV. In 2004, two studies by Sebat [Sebat et al., 2004] and Iafrate [Iafrate et al., 2004] indicated that intermediate-size Structural Variants , not visible at the karyotype level, represented a substantial amount of genetic variation, that had yet been unexplored. Both papers described the presence of common large CNVs in phenotypically normal individuals, opening a new field in genetic analysis. Several other studies followed in their steps, describing hundreds of insertion, deletion and inversion polymorphisms in both phenotypically normal and abnormal individuals. By 2006, it was already speculated that these types of variations contributed at least as much as SNPs and variable number of tandem repeats (both mini- and microsatellites), if not more, to interindividual genetic variability. It was also clear that these novel type of variants, especially CNVs, had the potential to affect gene dosage, and thus could be related to genetic disease, as was the case for the several variants involved in neurodevelopmental disorders. [Marshall et al., 2008] [Tyson et al., 2005] [Sebat et al., 2007] [De Vries et al., 2005] In fact, SV in human genomes has often been related to disease. In some cases, involving relatively rare SVs with large effects in rare disorders; in others, relatively common SVs have been shown to have smaller effects in complex diseases. On the contrary, it is a small proportion of the total number of known structural variants that has so far been related to disease [Escaramís et al., 2015]

2.1.2 Importance of Structural Variants

Structural variations (SVs) have received less attention in research compared to single nucleotide variations (SNVs) due to their inherent difficulty in identification. In theory, each type of SV produces a distinct pattern in the alignment of sequencing reads, allowing for inference of the underlying mutation. For instance, a deletion creates a gap in the alignment, indicating the absence of a specific sequence relative to a reference. However, in practice, the situation is much more intricate. Firstly, sequencing and mapping errors introduce noise that blurs the patterns. Unlike SNVs and smaller insertions/deletions, SVs can encompass a significant portion of a read or even exceed the read length, making mapping more challenging. Secondly, the patterns induced by different SV types can exhibit similarities, leading to difficulties in distinguishing between tandem duplications and novel insertions in genomic alignments, for example. Lastly, multiple SVs can overlap or be nested, resulting in complex mapping patterns that are far more intricate than when considering SVs individually. [Mahmoud et al., 2019]

In the past few years, the definition of structural variants and their breakpoints has been greatly improved with the use of next-generation sequencing technologies and the development of algorithms to detect structural variants more accurately in whole genome and exome data. [Escaramís et al., 2015]

2.1.3 Type of Structural Variant

The main types of structural variants considered here involve: translocations, where a segment of DNA changes its position, intra- or interchromosomally, without gain or loss of genetic material; inversions, defined as segments of DNA that are reversed in orientation from the rest of the chromosome, termed pericentric if they involve the centromere or paracentric otherwise; insertions of novel sequence with respect to a reference genome, including mostly mobile element insertions (MEIs); and CNVs, where a segment of DNA is present in a variable number of copies when compared to a reference genome. The latter include deletions, when there is a loss of genetic material with respect to the reference, and duplications of a given sequence, which can be inserted contiguously (in tandem) or elsewhere in the genome.

In addition, complex rearrangements might involve inversions or translocations flanked by CNV, or other combinations of events. A particular type of complex rearrangements are those formed by chromothripsis, a process whereby multiple classes of structural rearrangements occur simultaneously, generating multiple clustered deletion, translocation and inversion events. [Escaramís et al., 2015]

2.1.4 Detection of Structural Variants

The most well-studied types of structural variants are those involving changes in copy number of genetic material, as those are easier to detect with current technologies. Detection of balanced structural variants, aside from those large enough to be visible on a karyotype, generally involves the analysis of paired-end NGS data and the application of complex algorithms.

In 2004, the field of SV analysis was opened thanks to the development of array-CGH, which remains a robust method for the discovery of CNV. [Hupe et al., 2004] Array-CGH is based on the analysis of intensity ratios of the hybridization of two differentially dyed DNAs against the same target oligonucleotides. One of the disadvantages of array-CGH is that it detects imbalances between two individual genomes, and thus it cannot provide an absolute copy number. Array-CGH is also blind to balanced SV events.

More recently, the arrival of NGS and the rapid increase in its throughput have enabled the exploration of genomic SVs at an even finer scale. Initially, most NGS was performed with single reads (DNA is fragmented and sequenced from one end only). As technology developed, sequencing of paired-end reads (DNA is fragmented and both ends of the fragments are sequenced, with or without a non-sequenced stretch in between) became more common. Paired-end reads are better aligned to the reference and offer additional information for the detection of structural variants. This has allowed the detection of small CNVs, which were missed due to low resolution in array-CGH or SNP arrays, and it has enabled the genome-wide characterization of breakpoints for all classes of SVs. However, the characterization of structural variants from NGS is not straightforward, and many algorithms have been developed to try to gather the most accurate information. These algorithms follow one (or a combination) of four main strategies to identify SVs from NGS data: read depth, paired-end, split and clip reads and de novo assembly. Here I will briefly summarise the main features defining the four strategies:

(A) Read depth. Reads are aligned into the reference genome and when compared to diploid regions they show a reduced number of reads in a deleted region or higher read depth in a duplicated region.

(B) Paired reads. Pairs of sequence reads are mapped into the reference genome (from left to right). Different outcomes are expected depending on the presence or not of structural variants and their characteristics: (1) no SV, pairs are aligned into the correct order, correct orientation and spanned as expected based on the library's insert size; (2) deletion, the aligned pairs span far apart from that expected based on library insert size; (3) tandem duplication, read pairs are aligned in an unexpected order, where expected order means that the leftmost read should be aligned in the forward strand and the rightmost read in the reverse strand; (4) novel sequence insertion, the pairs are aligned closer from that expected based on library insert size; (5) inversion, read pairs are aligned in the wrong orientation, both reads align either in forward or reverse strand, and (6) read pairs mapped to different chromosomes.

(C) Split reads. Sequenced reads pointing to the same breakpoint are split at the nucleotide where the breakpoint occurs. The corresponding paired read is properly aligned to the reference genome.

(D) De novo assembly. Sample reads from novel sequence insertions are assembled without a reference sequenced genome. Each of these approaches has its own strengths and weaknesses, but none is capable of detecting the full spectrum of structural variants. All the listed tools are designed for the detection of structural variants from whole-genome sequencing data. However, most NGS studies performed to date have taken advantage of the reduced costs (both computing and economic) of Whole-Exome Sequencing. Exome sequencing focuses on protein-coding regions, which only cover a sparse 1 percent of the entire genome. This scenario, involving small and discontinuous target regions (exons), limits the detection capacity for structural variants. So far, only RD approaches have been successfully integrated from whole-genome sequencing to WES data. [Escaramís et al., 2015]

2.1.5 Structural Variants Pipeline

Structural variants calling is a complex process that includes a lot of steps. The first step is sequencing, in my case DNA sequencing. This process aims to determine the exact sequence of bases that compose the DNA that we want to analyze. There are different types of sequencing, in my case, the data were obtained through long-read sequencing. In this process is obtained as output a file that stores the raw data got from the analysed DNA. In my case, this process was done through ONT's nanopore. During this analysis, the DNA strand passes through the nanopore and each type of base generates a different change in electricity current. In that way is possible to distinguish each base. The output file of this process collects the raw data of electricity current changes. In order to transform these data in an exploitable format it's needed the next step. Indeed after DNA sequencing, there comes basecalling process. Base callers transform raw data into text format files where each base is stored using the corresponding letter. The next step of our pipeline is mapping/alignment of the reads against a reference genome, in my case Human Genome. That process aligns the sequenced reads, and tries to find the right position in the genome. Once done this is possible to run SV callers that take as inputs the alignment files and give as output a file where are stored all the differences that the DNA we analyzed has from the genome.

2.2 Data

In this project, I worked with several different types of file formats. The following is a brief description of the file formats I used in my project:

FAST5

The FAST5 format stores electrical signals detected by the nanopore. In these files we find the electrical changes that DNA strands generate pass through the nanopore. These fluctuations correspond to different bases in the DNA strand. These files are the starting point for my project.

FASTA

The FASTA format is a text-based format used for representing DNA sequences, commonly denoted by the file extensions ".fa" or ".fasta". The FASTA format is known for its simplicity. It starts with a header line, followed by lines of sequence data. The header line begins with the "¿" symbol, which signifies the sequence identifier, and it may also include additional optional descriptive information. The sequence lines themselves consist of characters that represent the nucleotide bases present in the sequence. Typically, each sequence line contains no more than 80 characters, but there is no restriction on the total number of lines. Each nucleotide base is encoded as a single character, with options including A, T, G, C, and N (used for undetermined bases), with the encoding being case-insensitive. [Zhang, 2016]

FASTQ

The FASTQ format is utilized for storing both the nucleotide sequence and its corresponding Phred quality scores, and it has become a widely adopted file format for sharing sequencing read data.

FASTQ files are plain text files with the extensions ".fq" or ".fastq" and can be directly viewed from the command line on computers operating on Unix/Linux systems. Within a FASTQ file, each sequence (representing a short read from next-generation sequencing) is defined by four lines of text:

The first line begins with the "@" character, followed by a sequence identifier and an optional description.

The second line consists of the raw sequence letters: A, T, G, C, and N (representing unknown bases).

The third line starts with the "+" symbol and may optionally include the same sequence identifier (along with any associated description) again. The "+" sign acts as a marker indicating the end of the sequence.

The fourth line contains the quality values for the sequence presented in the second line. It must contain the same number of symbols as there are letters in the sequence. [Zhang, 2016]

SAM and BAM

The BAM/SAM file plays a crucial role in the analysis of next-generation sequencing data because the raw sequence reads obtained from sequencers lack genomic position information and need to be mapped or aligned to a known reference genome. The output of the mapping or alignment process is represented in the form of BAM/SAM files, which serve as inputs for various downstream analyses, including feature counting and variant calling.

SAM, which stands for Sequence Alignment/Map format, is a versatile alignment format designed to store read alignments against reference sequences. It supports both short and long reads, with a capacity of up to 128 megabases, and is compatible with diverse sequencing platforms. SAM files typically bear the file extension ".sam" and are structured as tab-delimited text files that encompass data pertaining to sequence alignment.

On the other hand, BAM files represent the binary version of SAM files and typically carry the file extension ".bam". Both BAM and SAM files contain identical information, and SAMtools software can be employed to easily convert a BAM file into a SAM file. [Zhang, 2016]

VCF

The VCF (Variant Call Format) format was initially introduced by the 1000 Genomes Project to facilitate the storage of prevalent types of genomic sequence variations, including SNPs (single nucleotide polymorphisms) and small INDELs (insertions/deletions), along with associated annotations. The VCF file is a text-based file that comprises meta-information (header) lines and data lines.

Most VCF files consist of both header lines and data lines. The header lines begin with "##" and follow an "ID=value" format. The first header line always specifies the version of the VCF format. Subsequent header lines, starting with "##INFO=", "#FILTER=", and "##FORMAT", define the name, length, value type, and description of each element within the relevant fields (columns) of each data line.

In summary, the VCF format employs a text file structure, encompassing metadata and actual variant data. The header lines provide essential information and descriptions, while the subsequent data lines contain the specific variant call data. [Zhang, 2016]

3 Methods

3.1 Tools

3.1.1 Cluster

The whole project has been developed through a cluster. Due to the large amount of data that i had to work with, the research center allowed me to use its cluster. I learnt how to work with it through SLURM, a job scheduler used by their computer cluster. After building up my snakemake script i ran it in the cluster. The cluster features two Nvidia A100 GPUs, so ran it using the GPU partition. That was especially useful to basecall the FAST5 data, because base-calling was the most time-consuming task that i had to do.

3.1.2 Snakemake

Snakemake, a widely used workflow manager system, written in Python and inspired by GNU Make. It allows for the composition of workflows based on a graph of rules whose execution is triggered by the presence, absence, or modification of expected files and directories. [Strozzi et al., 2019] [Mölder et al., 2021]

I used Snakemake to build a pipeline that allowed me to run several softwares subsequently, each one using the outputs of the previous one. I also used several parameters in each rule, especially "conda" and "benchmarks". In "Rules Parameter" 3.2.3 section will be explained more in depth their crucial roles in the pipeline.

3.1.3 Conda

Conda is a cross-platform open-source system for managing packages and environments. It operates on Windows, macOS, and Linux, facilitating the swift installation, execution, and updating of packages and their dependencies. Additionally, Conda provides effortless creation, storage, loading, and switching between environments on your local computer. [Con,] The whole project is based on Conda environments. Conda was crucial for my work because is a fast way to use several tools, that are easily available through it. I used a vast amount of software, downloaded through Conda, more in depth through Bioconda and Conda-forge channels, where are stored the majority of the Bioinformatic's tools. I created separate environments for each tool, to avoid dependencies problems, that are frequent using Conda. I manually perfomed each step of the pipeline individually, creating a dedicated environment for that part of the final work. Only after a successful run of the single tool i implemeted the corrispondent rule in Snakemake, replicating the separated environments due to Snakemake's parameter "Conda".

3.1.4 Guppy

Guppy is the basecaller developed by ONT. The current version of ONT's Guppy basecaller performs well overall, with good accuracy and fast performance Basecalling was performed using the Guppy 2.3.1 software. [Wick et al., 2019] I downloaded Guppy from the Oxford Nanopore Technologies website and i used the GPU version of it to accelerate the process.

3.1.5 Nanoplot

NanoPlot is a tool with various visualizations of sequencing data in bam, cram, fastq, fasta, or platform-specific TSV summaries, mainly intended for long-read sequencing from Oxford Nanopore Technologies and Pacific Biosciences. Nanoplot is contained in NanoPack. now offers tools for evaluating large populations with implementations in a more performant programming language, focusing on features relevant to long-read sequencing. [De Coster and Rademakers, 2023]

3.1.6 Minimap2

Minimap2 is a versatile alignment software designed for mapping DNA or long mRNA sequences onto a vast reference database. It is capable of handling precise short reads with a minimum length of 100bp, genomic reads of at least 1kb length with an error rate of around 15%, full-length noisy Direct RNA or cDNA reads, as well as assembly contigs or closely related full chromosomes spanning hundreds of megabases.

Minimap2 employs split-read alignment, incorporates a concave gap cost model to accommodate lengthy insertions and deletions, and introduces novel heuristics to minimize false alignments. Compared to popular short-read mappers with similar accuracy, Minimap2 demonstrates a 3−4 times faster performance, while outperforming specialized aligners for long-read genomic or cDNA mapping by at least 30 times in terms of speed and accuracy. [Li, 2018]

3.1.7 Samtools

SAMtools is a library and software package for parsing and manipulating alignments in the SAM/BAM format. In my case i used Samtools to convert SAM file into BAM file and after that i used again Samtools to sort and index the alingments.

3.1.8 Sniffles2

A fast structural variant caller for long-read sequencing, Sniffles2 accurately detect SVs on germline, somatic and population-level for PacBio and Oxford Nanopore read data. Through Sniffles2 i got a vcf file where all the structural variations are stored. Sniffles just require the alingment file to run. [Sedlazeck et al., 2018]

3.1.9 CuteSV

CuteSV is an adaptable approach for detecting structural variations (SVs) based on read alignments, offering several advantageous features. Firstly, cuteSV exhibits superior SV detection performance compared to state-of-the-art SV callers. It demonstrates particularly higher sensitivity for low coverage datasets without compromising accuracy. Secondly, cuteSV supports datasets generated by mainstream long-read sequencing platforms, accommodating various error rates, and effectively detecting diverse types of SVs, including deletions, insertions, duplications, inversions, and translocations. Thirdly, cuteSV showcases faster or comparable speed to existing approaches while utilizing lower RAM resources. [Jiang et al., 2020]

3.1.10 BCFtools

BCFtools is a program for variant calling and manipulating files in the Variant Call Format (VCF) and its binary counterpart BCF. [Danecek et al., 2021] I used BCFtools to extract information from my vcf files in order to understand the results of my work.

3.2 Pipeline

3.2.1 Introduction

My project, aims to detect structural variations in a patient's DNA affected by multiple myeloma. In order to do that i created a pipeline using Snakemake that consist in a sequence of software that combined drove me from the original data, the whole genome, to a vcf file, where are stored all the structural variations. The starting data consists in FAST5 files obtained from Altum Sequencing, sequencing the whole genome of a multiple myeloma patient. They performed twice the same sequencing on the same genome using two different flow cells. Starting from those files i performed a basecalling on them through guppybasecaller and subsequently i used Nanoplot to do quality control. The next step was to concatenate all the fastq files of each sequencing in just one big fastq file. I thereafter i used Minimap2 on the latter to align the reads contained in the fastq to/with the reference genome, obtaining a sam file. Then i converted it into a bam file in order to sort and index it. After all this i got all the necessary files to run the SV callers, which are Sniffles2 and CuteSV. In the end, i got different vcf files that i was able to analyze. In order to extract some useful information from those files i used BCFtools to filter and explore the data. I created a bash script that allowed me to run several BCFtools commands and in that way it was also easy to repeat this process for all the vcf files that i got from the different callers that i used. The bash script collected all the data in the Analyses.tsv file setting in different columns the callers name and in the rows all the extracted information. In that way was easier to compare the results. Everything has been possible owing to CNIO, that allowed me to take advantage of its structures and resources. [CNI,]

3.2.2 Data

I used two sets of FAST5 files, obtained by Altum Sequencing after sequencing the whole genome of a single Myeloma Cancer patient with two different flow cells.Altum Sequencing also provided a fastq file but after a while i found out that there were problems in those files, so i did the basecalling by myself and then everything worked.

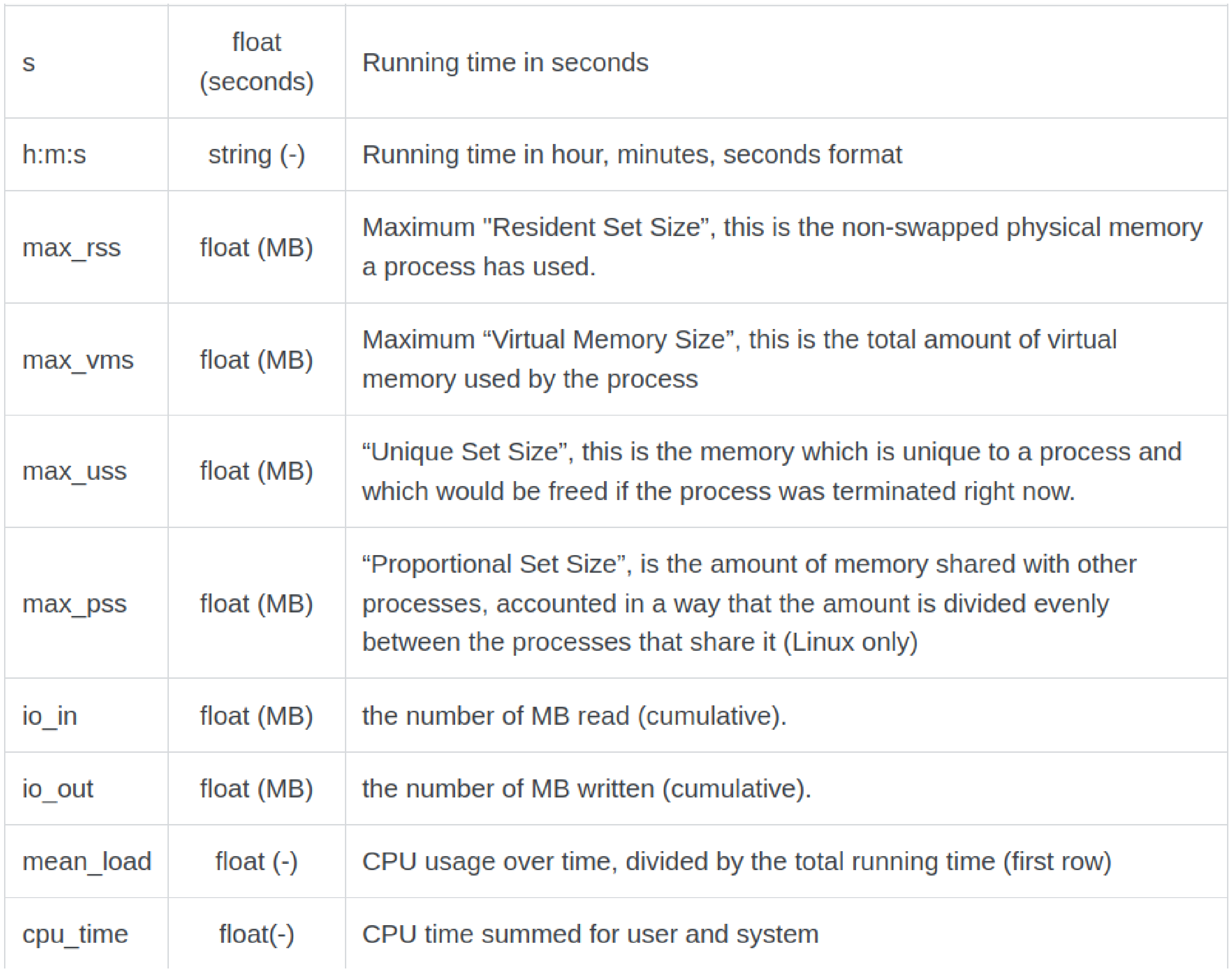
3.2.3 Rules parameters

Conda

This Snakemake's feature allows to specify for each rule an environment that would be created in order to run each rule in a well-defined environment. This feature fixes a lot of dependencies problems that may occur installing all the packages in the same environment. In order to make it clearer possible i created a separate environment for each rule. They are stored in the envs folder and each rule in the conda parameter has specified the .yaml file's path.

Benchmark

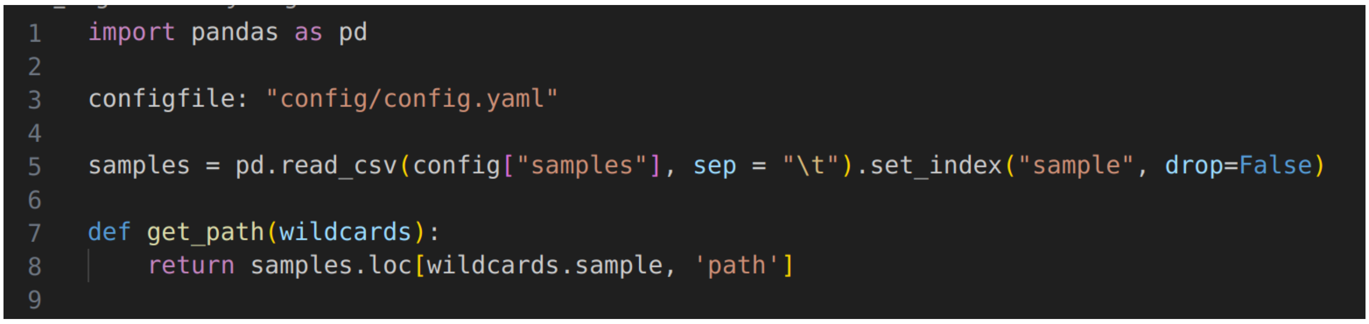
This Snakemake's parameter is very useful to understand how many resources each rule needs. Every rule has a separate .tsv file where at the first run snakemake stores the resources used by the rule. That allows you to adjust the amount of time and memory given to the rule. That's really important because in a shared cluster is crucial to not waste resources. These following table explains the stats described in each benchmark file.



s	float (seconds)	Running time in seconds
h:m:s	string (-)	Running time in hour, minutes, seconds format
max_rss	float (MB)	Maximum "Resident Set Size", this is the non-swapped physical memory a process has used.
max_vms	float (MB)	Maximum "Virtual Memory Size", this is the total amount of virtual memory used by the process
max_uss	float (MB)	"Unique Set Size", this is the memory which is unique to a process and which would be freed if the process was terminated right now.
max_pss	float (MB)	"Proportional Set Size", is the amount of memory shared with other processes, accounted in a way that the amount is divided evenly between the processes that share it (Linux only)
io_in	float (MB)	the number of MB read (cumulative).
io_out	float (MB)	the number of MB written (cumulative).
mean_load	float (-)	CPU usage over time, divided by the total running time (first row)
cpu_time	float(-)	CPU time summed for user and system

3.2.4 Configuration

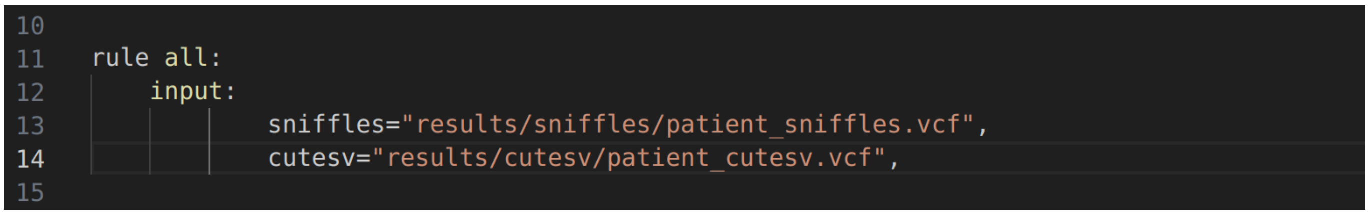
In the first part of the code, I imported pandas' library. Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. [Pan,] After it, I declared the configfile where is stored a tsv file in which are collected the two sample paths that I used in my pipeline. I also declared the variable samples, that read the tsv file, storing the paths before mentioned. Subsequently, I defined the function get_path that I used in Guppy's rule. This function reads the sample variables and returns the path of the corresponding sample. Through this rule, I was able to use the Snakemake pipeline for all the samples that I have running just one snakemake command. In this way, the pipeline is more scalable and in order to perform it on further samples it's only needed to add them to the sample.tsv file.



```python
import pandas as pd

configfile: "config/config.yaml"

samples = pd.read_csv(config["samples"], sep = "\t").set_index("sample", drop=False)

def get_path(wildcards):
    return samples.loc[wildcards.sample, 'path']

```

3.2.5 Rule all

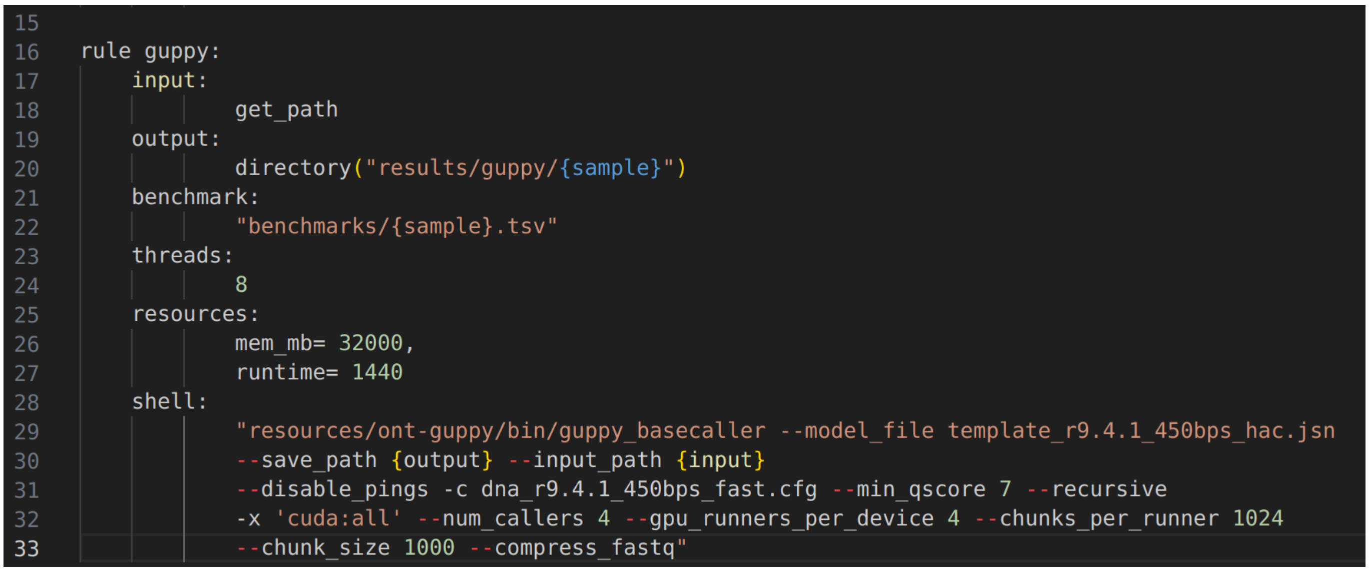
In this rule, I specified the final output file that the pipeline has to create. That's why I specified in the input the Sniffles' and CuteSV's vcf files. This rule calls the rules that have those two files as outputs. CuteSV's rule and Sniffles' rule do the same with their inputs calling the rules that as them as output. This process reaches in the end, the "first" rule that Snakemake has to perform, which in my case is Guppy's rule. Then all the rules are run subsequently until getting the files specified in the rule all.



```python

rule all:
    input:
            sniffles="results/sniffles/patient_sniffles.vcf",
            cutesv="results/cutesv/patient_cutesv.vcf",

```

3.2.6 Guppy

In the first rule, I took as inputs the folders containing the two sets of FAST5 files, from the different sequencing. Guppy created as output two folders each one storing the outputs of one set. Called reads were classified as either pass or fail depending on their mean quality score. In each of these folders, we can find different files in which there are also pass and fail folders. In the pass folder, we can find all the reads with good quality, and in fail we can find all the reads that Guppy considers with bad quality. Between the outputs files, we can also find summary_sequencing.txt that gives us all the stats regarding all the reads.



```python

rule guppy:
    input:
            get_path
    output:
            directory("results/guppy/{sample}")
    benchmark:
            "benchmarks/{sample}.tsv"
    threads:
            8
    resources:
            mem_mb= 32000,
            runtime= 1440
    shell:
            "resources/ont-guppy/bin/guppy_basecaller --model_file template_r9.4.1_450bps_hac.jsn
            --save_path {output} --input_path {input}
            --disable_pings -c dna_r9.4.1_450bps_fast.cfg --min_qscore 7 --recursive
            -x 'cuda:all' --num_callers 4 --gpu_runners_per_device 4 --chunks_per_runner 1024
            --chunk_size 1000 --compress_fastq"
```

Guppy rule

Guppy's parameter description:

−−save_path output = Output path

−−input_path input = Input path

−−disable_pings = Disable the transmission of telemetry pings. By default, MinKNOW automatically sends experiment performance data to Nanopore.

−*c* dna_r9.4.1_450bps_fast.cfg = Configuration file, regarding flow cell type and kit type. In my case Flow Cell type: FLO-MIN106 (R9.4.1), kit type: SQK-LSK110

−−min_qscore 7 = Minimum acceptable qscore for a read to be filtered into the PASS folder

−−recursive = Search for input file recursively

−x 'cuda:all' = GPU device to use in order to accelerate basecalling.

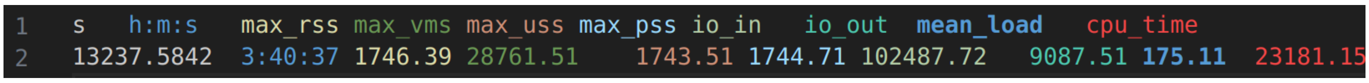
−−num_callers 4 = Number of parallel basecallers to create

−−gpu_runners_per_device 4 = Number of runners per GPU device

−−chunks_per_runner 1024 = Maximum chunks per runner

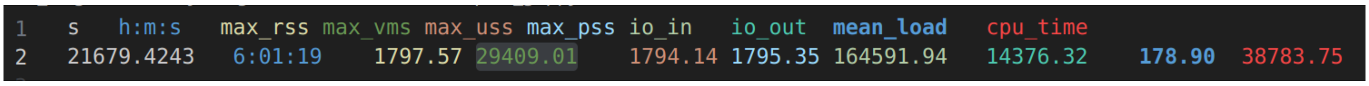
−−chunk_size 1000 = Stride intervals per chunk

−−compress_fastq = Compress fastq output files with gzip



```
s          h:m:s    max_rss max_vms  max_uss  max_pss io_in     io_out  mean_load  cpu_time
13237.5842 3:40:37  1746.39 28761.51   1743.51 1744.71 102487.72   9087.51 175.11  23181.15
```

Guppy used resources sample 1

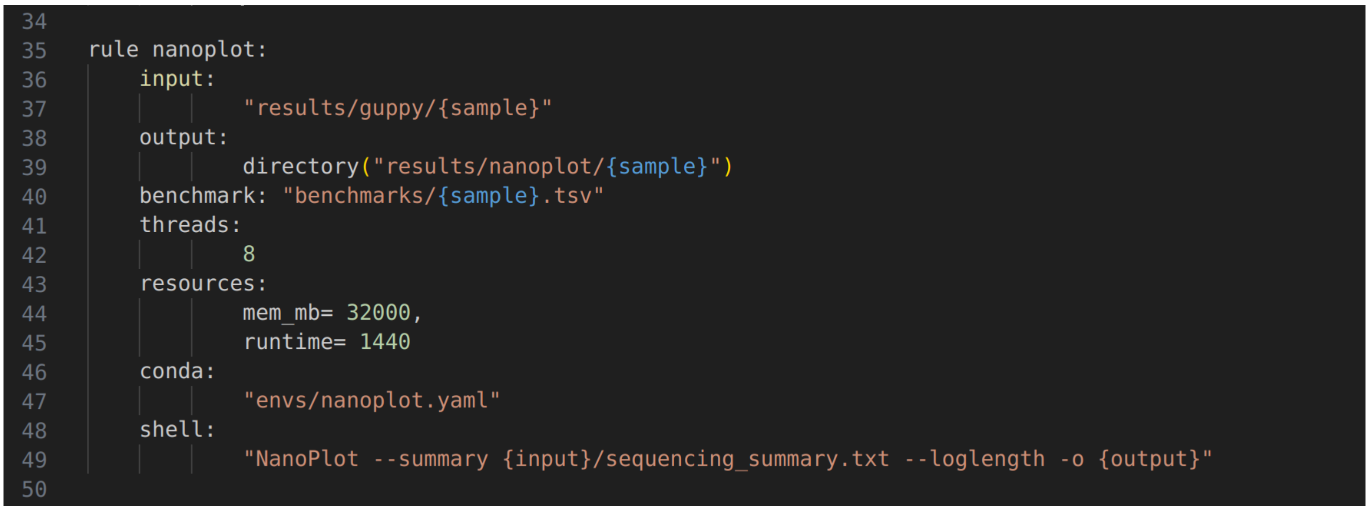


```
s          h:m:s    max_rss max_vms max_uss max_pss io_in     io_out   mean_load  cpu_time
21679.4243  6:01:19     1797.57 29409.01    1794.14 1795.35 164591.94 14376.32     178.90  38783.75
```

Guppy used resources sample 2

3.2.7 Nanoplot

In this rule, I used Nanoplot to do quality control over Guppy's outputs. Nanoplot as different ways to perform it, using different input files. I chose to use Guppy's summary.txt as input. I chose to use just the pass reads, so the reads with good quality. We can see that 90% of the reads has a Q7 plus quality, which means that 7 was the threshold that Guppy set to select pass and fail reads. Within the remaining 10%, half of them has a Q5 plus quality. The mean read quality is 9.4 while the median read quality is 11.4, over a 1,190,042.0 reads with a mean length of 7,717.6.



```

rule nanoplot:
    input:
            "results/guppy/{sample}"
    output:
            directory("results/nanoplot/{sample}")
    benchmark: "benchmarks/{sample}.tsv"
    threads:
            8
    resources:
            mem_mb= 32000,
            runtime= 1440
    conda:
            "envs/nanoplot.yaml"
    shell:
            "NanoPlot --summary {input}/sequencing_summary.txt --loglength -o {output}"

```

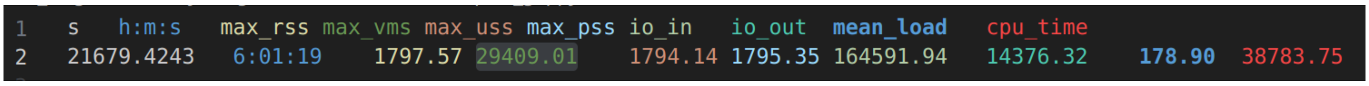
Nanoplot rule

Nanoplot's parameter description:

−−summary input/sequencing_summary.txt = Selection of Guppy's summary file as input.

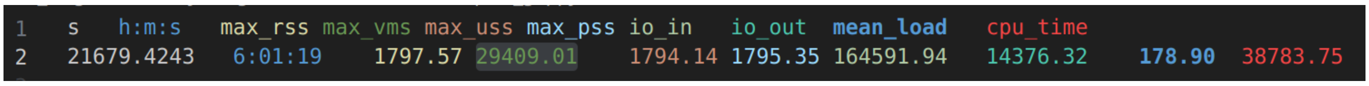
−−loglength = Additionally show logarithmic scaling of lengths in plots

−o output = Output path



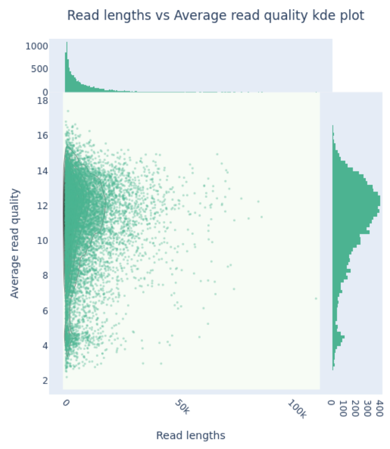
```
s    h:m:s    max_rss max_vms max_uss max_pss io_in    io_out   mean_load    cpu_time
21679.4243    6:01:19     1797.57 29409.01     1794.14 1795.35 164591.94    14376.32     178.90   38783.75
```

Nanoplot's used resources sample 1

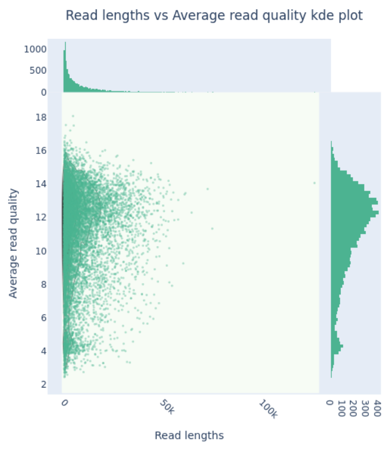


```
s    h:m:s    max_rss max_vms max_uss max_pss io_in    io_out   mean_load    cpu_time
21679.4243    6:01:19     1797.57 29409.01     1794.14 1795.35 164591.94    14376.32     178.90   38783.75
```

Nanoplot's used resources sample 2

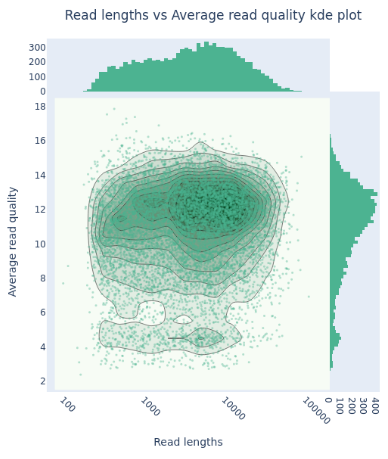


[Sample 1]

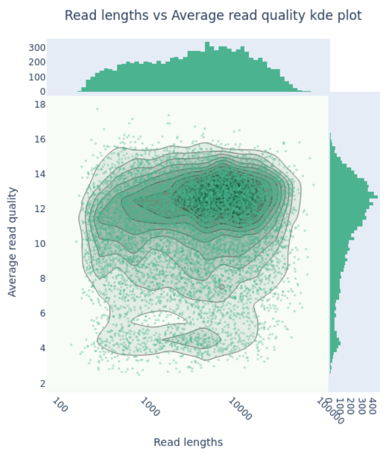


[Sample 2]

Figure 3.1: Lenght vs Quality Scatter Plot Comparison Sample1 and Sample 2

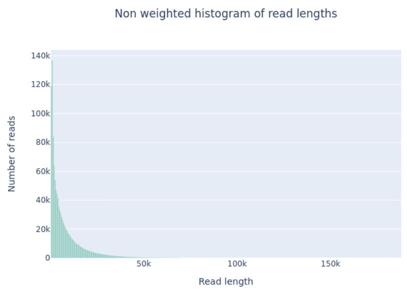


[Sample 1]

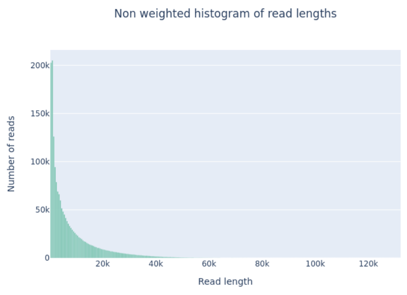


[Sample 2]

Figure 3.2: Lenght vs Quality Scatter Plot After Log Transformation Comparison Sample 1 and Sample 2

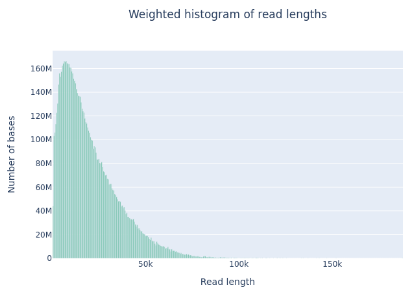


[Sample 1]

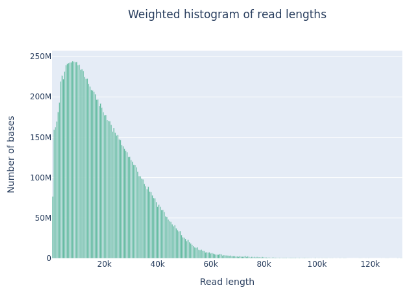


[Sample 2]

Figure 3.3: Lenght vs Quality Scatter Plot After Log Transformation Comparison Sample 1 and Sample 2

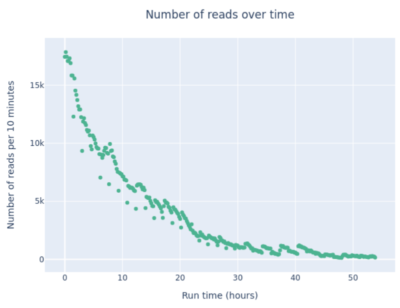


[Sample 1]

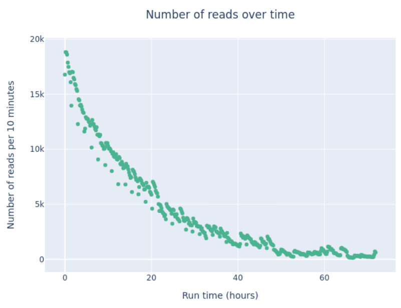


[Sample 2]

Figure 3.4: Weighted Histogram Read Lenaght Comparison Sample 1 and Sample 2

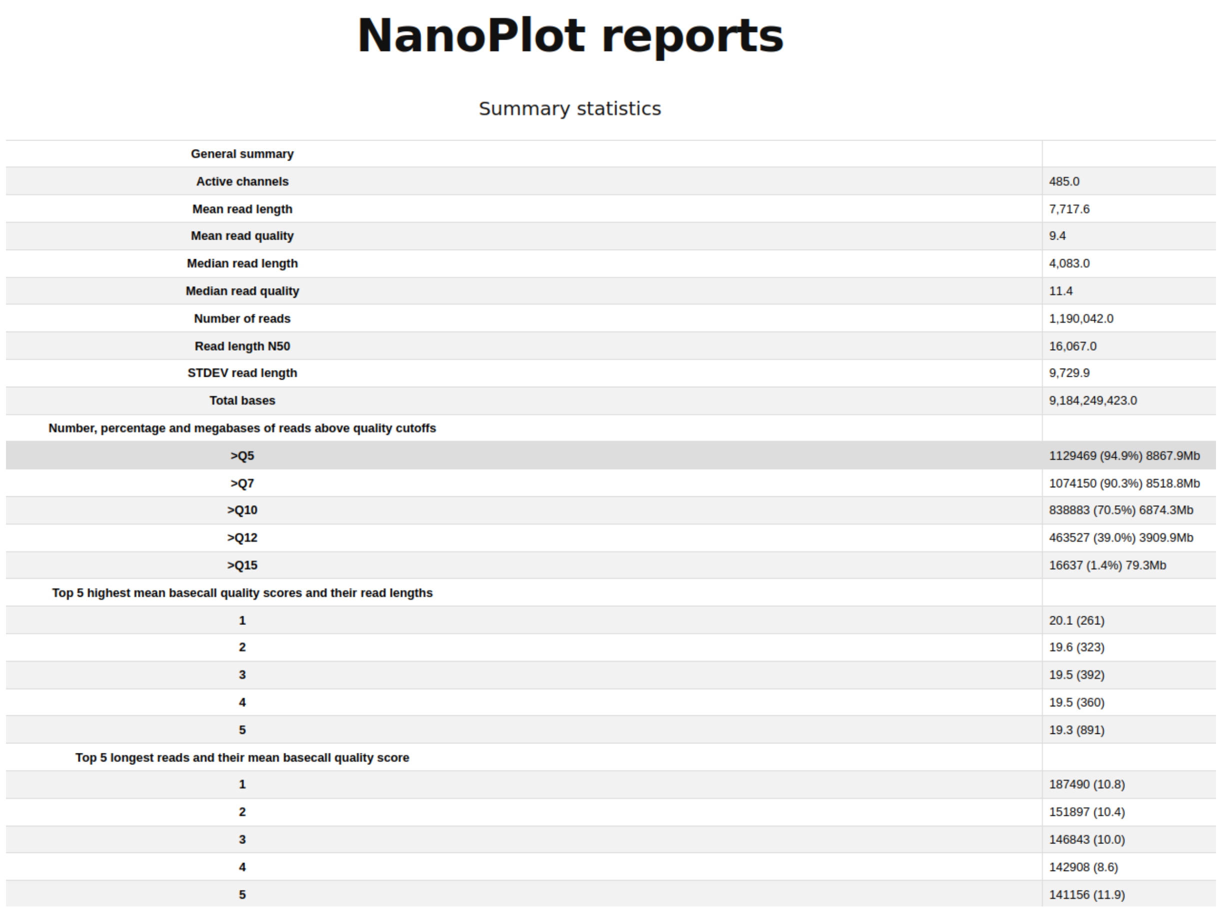


[Sample 1]



[Sample 2]

Figure 3.5: Number of reads over time Comparison Sample 1 and Sample 2

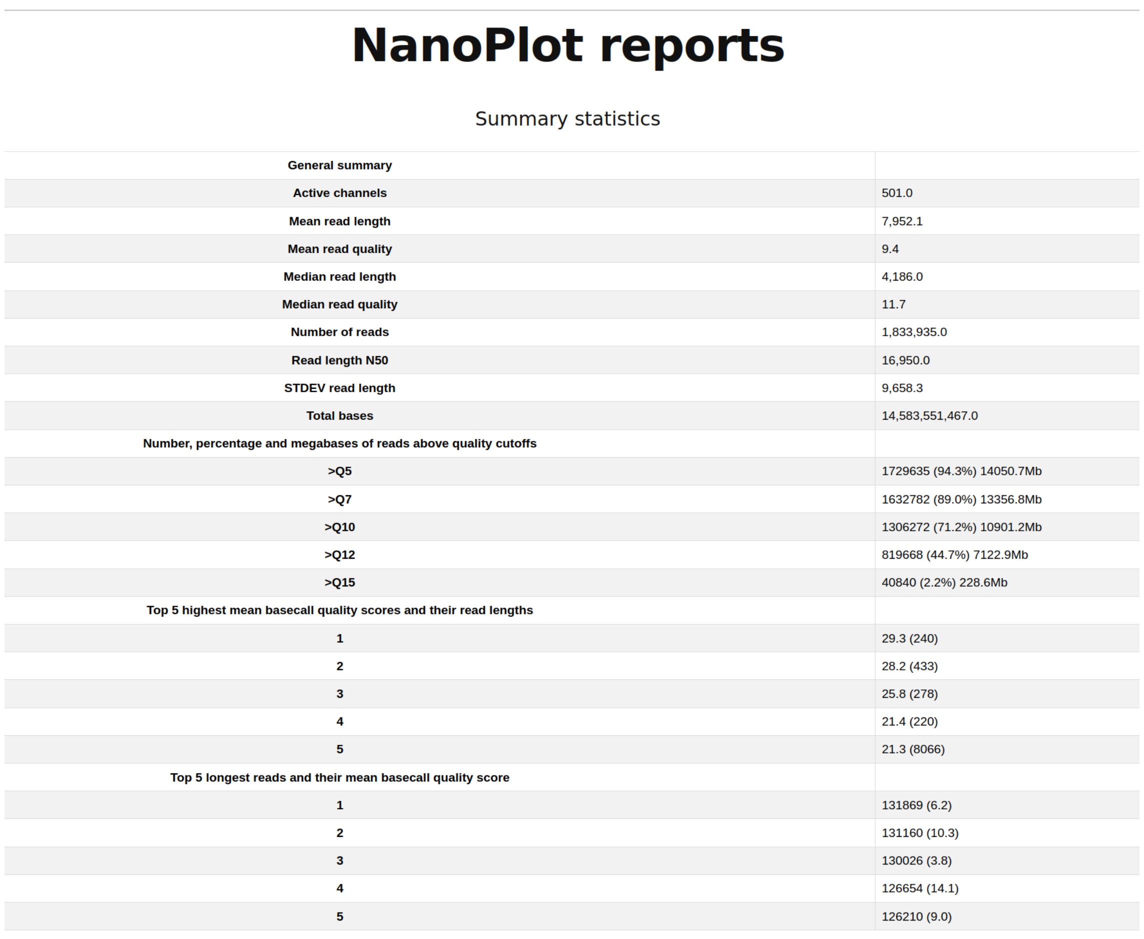


NanoPlot reports

Summary statistics

General summary	
Active channels	485.0
Mean read length	7,717.6
Mean read quality	9.4
Median read length	4,083.0
Median read quality	11.4
Number of reads	1,190,042.0
Read length N50	16,067.0
STDEV read length	9,729.9
Total bases	9,184,249,423.0
Number, percentage and megabases of reads above quality cutoffs	
>Q5	1129469 (94.9%) 8867.9Mb
>Q7	1074150 (90.3%) 8518.8Mb
>Q10	838883 (70.5%) 6874.3Mb
>Q12	463527 (39.0%) 3909.9Mb
>Q15	16637 (1.4%) 79.3Mb
Top 5 highest mean basecall quality scores and their read lengths	
1	20.1 (261)
2	19.6 (323)
3	19.5 (392)
4	19.5 (360)
5	19.3 (891)
Top 5 longest reads and their mean basecall quality score	
1	187490 (10.8)
2	151897 (10.4)
3	146843 (10.0)
4	142908 (8.6)
5	141156 (11.9)

Figure 3.6: Nanoplot Report Sample 1


NanoPlot reports

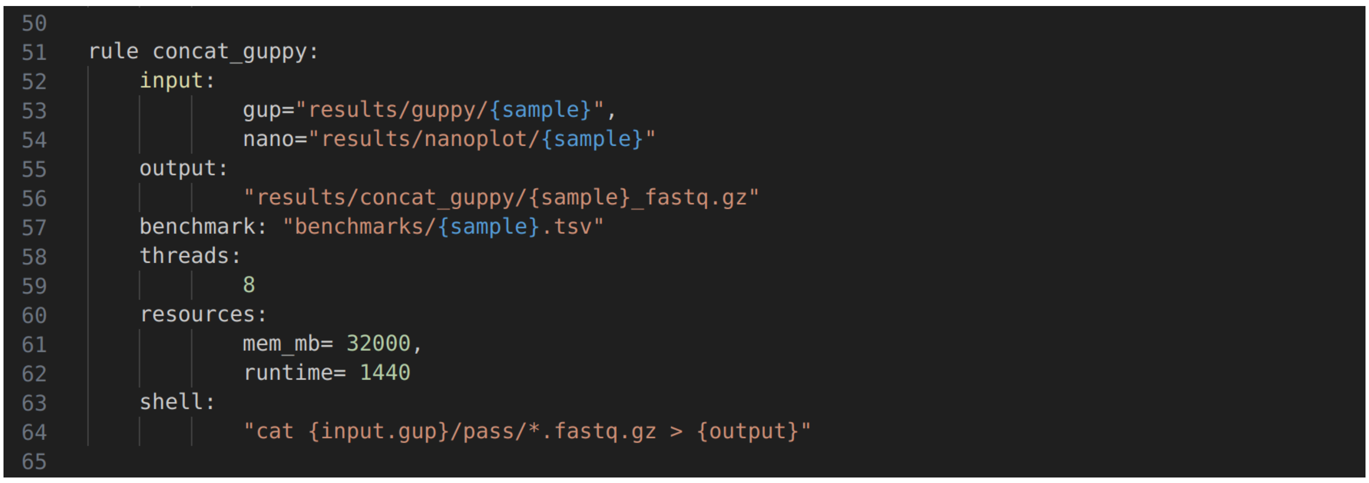
Summary statistics

General summary	
Active channels	501.0
Mean read length	7,952.1
Mean read quality	9.4
Median read length	4,186.0
Median read quality	11.7
Number of reads	1,833,935.0
Read length N50	16,950.0
STDEV read length	9,658.3
Total bases	14,583,551,467.0
Number, percentage and megabases of reads above quality cutoffs	
>Q5	1729635 (94.3%) 14050.7Mb
>Q7	1632782 (89.0%) 13356.8Mb
>Q10	1306272 (71.2%) 10901.2Mb
>Q12	819668 (44.7%) 7122.9Mb
>Q15	40840 (2.2%) 228.6Mb
Top 5 highest mean basecall quality scores and their read lengths	
1	29.3 (240)
2	28.2 (433)
3	25.8 (278)
4	21.4 (220)
5	21.3 (8066)
Top 5 longest reads and their mean basecall quality score	
1	131869 (6.2)
2	131160 (10.3)
3	130026 (3.8)
4	126654 (14.1)
5	126210 (9.0)

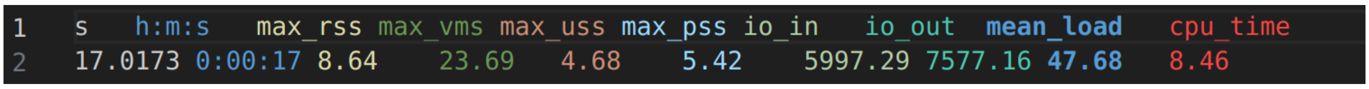
Figure 3.7: Nanoplot Report Sample 2

3.2.8 Concat Guppy

In this rule I concatenate all the compressed fastqs from each folder in one big compressed fastq file obtaining two fastqs, one for each set of data

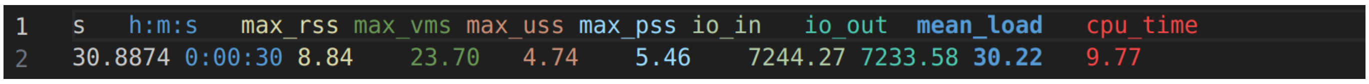

```

rule concat_guppy:
    input:
        gup="results/guppy/{sample}",
        nano="results/nanoplot/{sample}"
    output:
        "results/concat_guppy/{sample}_fastq.gz"
    benchmark: "benchmarks/{sample}.tsv"
    threads:
        8
    resources:
        mem_mb= 32000,
        runtime= 1440
    shell:
        "cat {input.gup}/pass/*.fastq.gz > {output}"

```

Concat guppy rule



```
s       h:m:s   max_rss max_vms max_uss max_pss io_in   io_out  mean_load  cpu_time
17.0173 0:00:17 8.64    23.69   4.68    5.42    5997.29 7577.16 47.68      8.46
```

Concat guppy's used resources sample 1

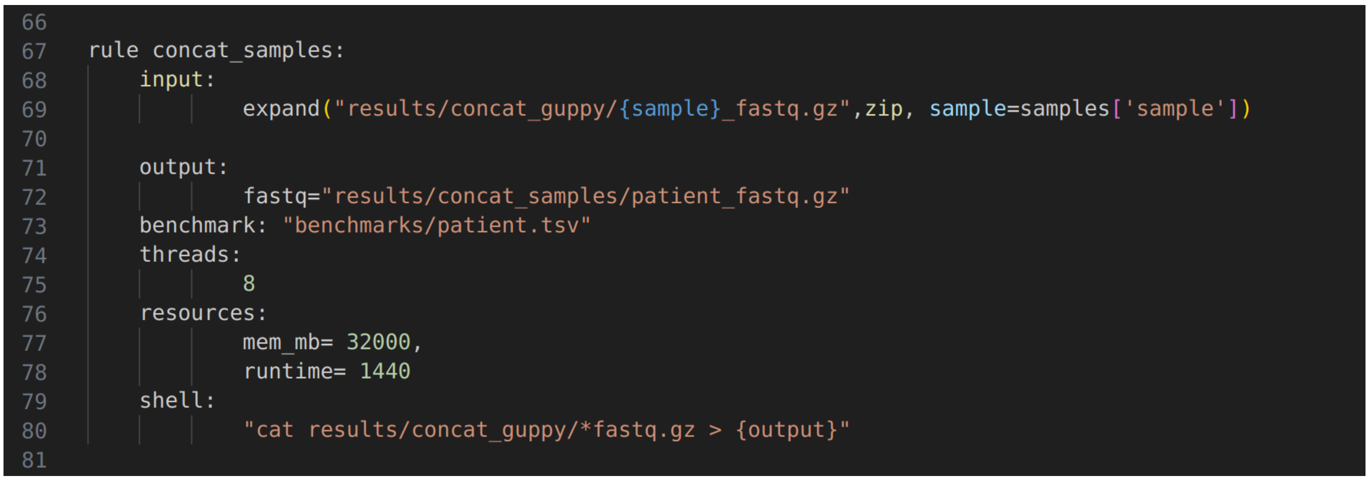


```
s       h:m:s   max_rss max_vms max_uss max_pss io_in   io_out  mean_load  cpu_time
30.8874 0:00:30 8.84    23.70   4.74    5.46    7244.27 7233.58 30.22      9.77
```

Concat guppy's used resources sample 2

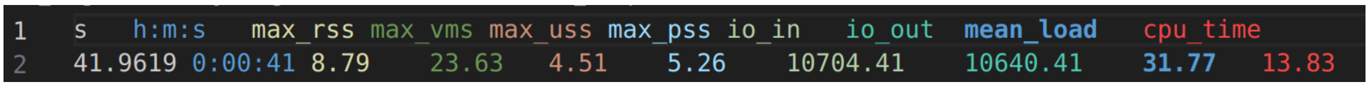
3.2.9 Concat Samples

In this rule I concatenate the two fastqs created in the previous rule, creating a single fastq that stores the basecalling of both the sequencing



```

rule concat_samples:
    input:
        expand("results/concat_guppy/{sample}_fastq.gz",zip, sample=samples['sample'])

    output:
        fastq="results/concat_samples/patient_fastq.gz"
    benchmark: "benchmarks/patient.tsv"
    threads:
        8
    resources:
        mem_mb= 32000,
        runtime= 1440
    shell:
        "cat results/concat_guppy/*fastq.gz > {output}"

```

Concat samples rule



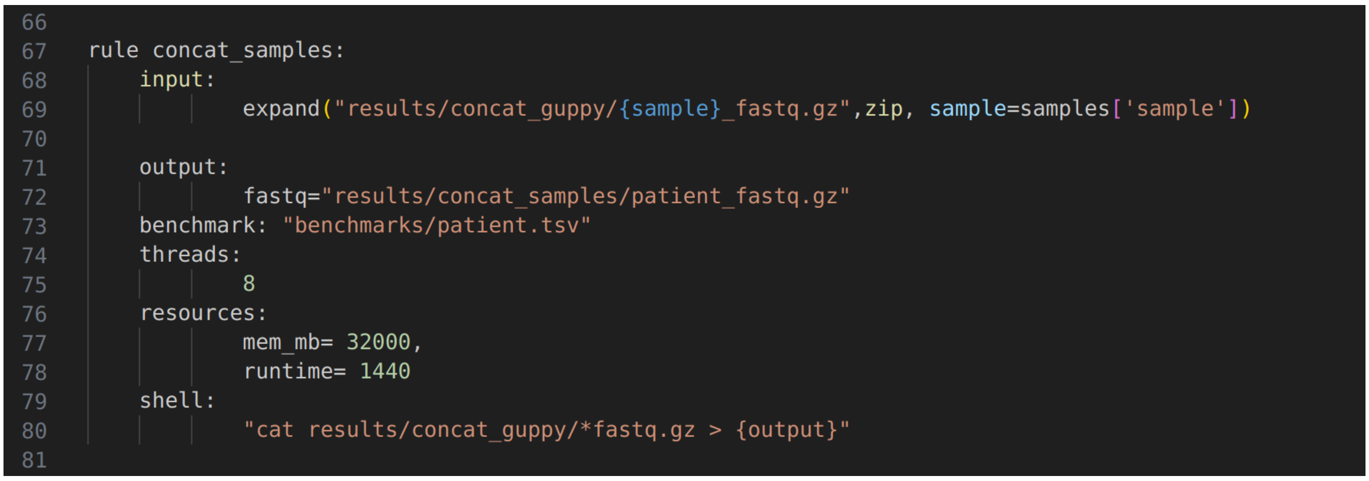
```
s       h:m:s   max_rss max_vms max_uss max_pss io_in    io_out    mean_load  cpu_time
41.9619 0:00:41 8.79    23.63   4.51    5.26    10704.41 10640.41  31.77      13.83
```

Concat samples' used resources

3.2.10 Minimap2

Minimap2 is a fast general-purpose alignment program to map DNA or long mRNA sequences against a large reference database.

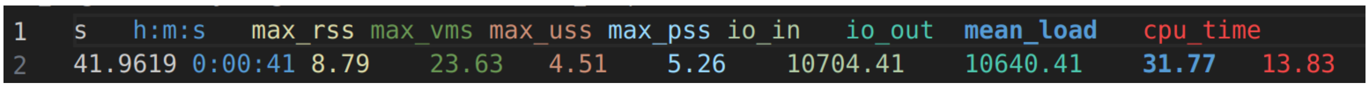
Minimap2 requires as inputs a fastq file and a reference genome. We are using "Genome sequence, primary assembly (GRCh38)" fasta file as reference genome and the output from concat_samples rule as fastq files. Minimap2 aligns the reads contained in the fastq against the reference genome, giving as output a sam file.



```

rule concat_samples:
    input:
        expand("results/concat_guppy/{sample}_fastq.gz",zip, sample=samples['sample'])

    output:
        fastq="results/concat_samples/patient_fastq.gz"
    benchmark: "benchmarks/patient.tsv"
    threads:
        8
    resources:
        mem_mb= 32000,
        runtime= 1440
    shell:
        "cat results/concat_guppy/*fastq.gz > {output}"

```

Minimap2 rule

Minimap2's parameter description:
−a = output in the SAM format
−x map−ont = pre-setting, Nanopore vs reference mapping



```
s       h:m:s   max_rss max_vms max_uss max_pss io_in    io_out   mean_load  cpu_time
41.9619 0:00:41 8.79    23.63   4.51    5.26    10704.41 10640.41 31.77      13.83
```

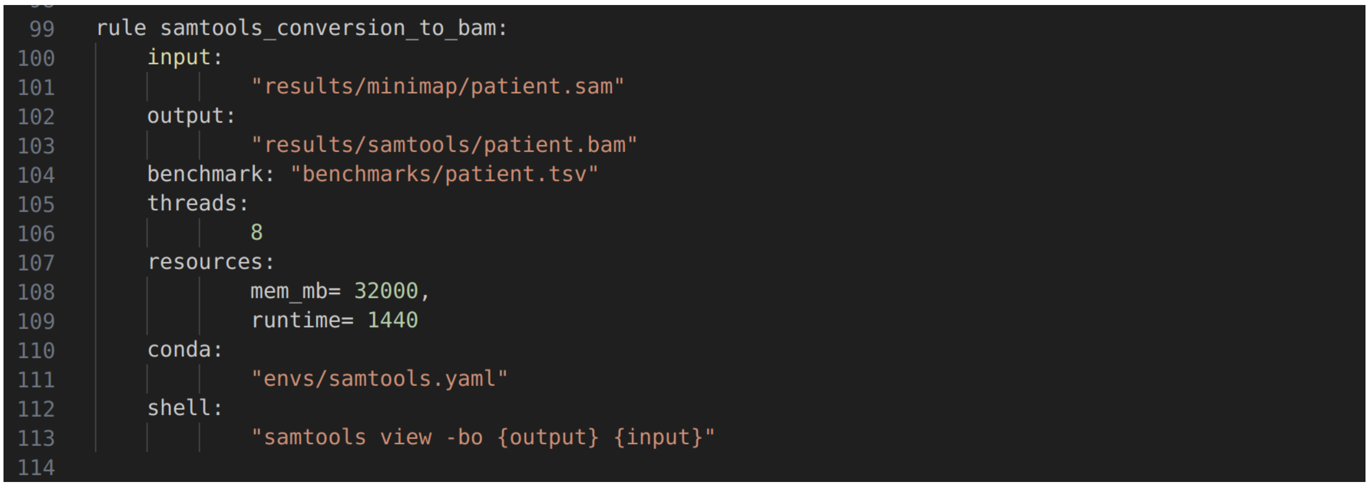
Minimap2's used resources

3.2.11 Samtools

SAMtools is a library and software package for parsing and manipulating alignments in the SAM/BAM format. In my case, I used Samtools to convert SAM file into BAM file and after that I used again Samtools to sort and index the alignments.

Samtools conversion to BAM

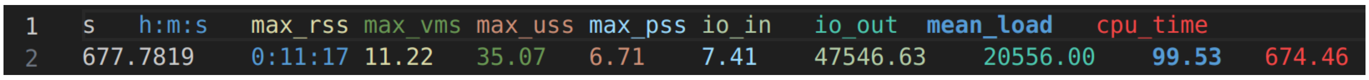
In this rule I converted the Minimap2 SAM output file into a BAM file, using the view parameter. This process is needed to improve performance. SAM file in fact is a human readable format while BAM is its binary version.



```
rule samtools_conversion_to_bam:
    input:
            "results/minimap/patient.sam"
    output:
            "results/samtools/patient.bam"
    benchmark: "benchmarks/patient.tsv"
    threads:
            8
    resources:
            mem_mb= 32000,
            runtime= 1440
    conda:
            "envs/samtools.yaml"
    shell:
            "samtools view -bo {output} {input}"

```

Samtools conversion to BAM rule

Samtools' conversion parameter description:
−bo output = set BAM as output format and specify output path

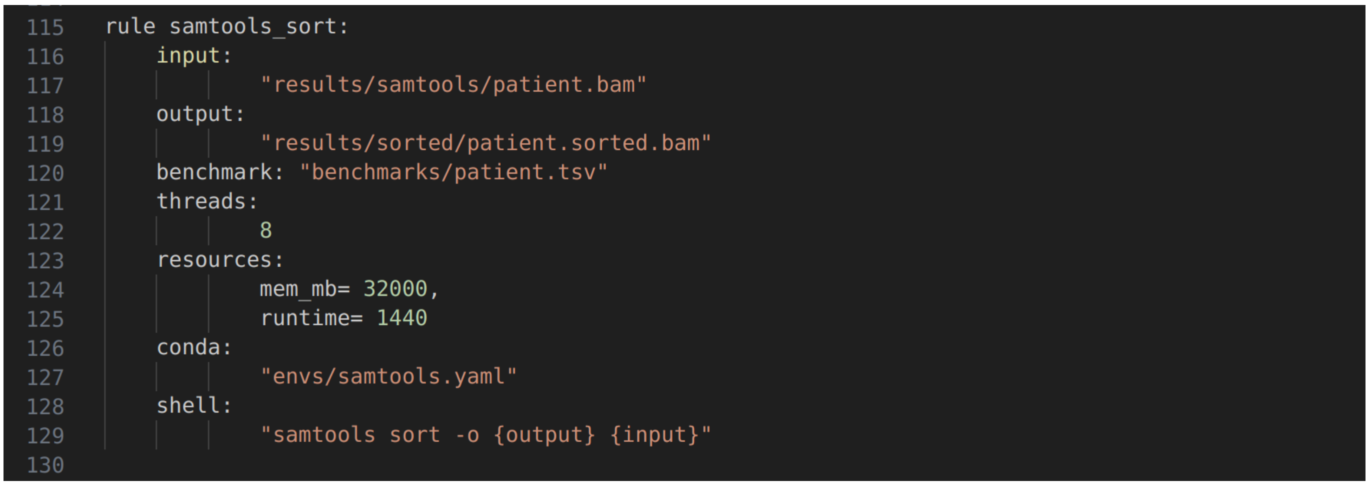


```
s        h:m:s   max_rss max_vms max_uss max_pss io_in    io_out   mean_load  cpu_time
677.7819 0:11:17 11.22   35.07   6.71    7.41    47546.63 20556.00 99.53      674.46
```

Samtools' conversion to BAM used resources

Samtools sort

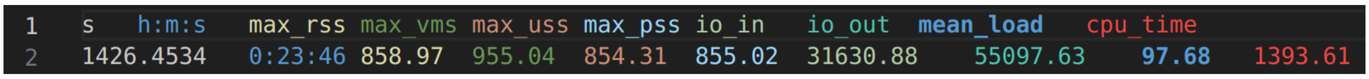
In this rule, I used the sorting tools provided by Samtools to sort the reads by coordinates. This process is useful to make the variant calling process faster.



```
rule samtools_sort:
    input:
            "results/samtools/patient.bam"
    output:
            "results/sorted/patient.sorted.bam"
    benchmark: "benchmarks/patient.tsv"
    threads:
            8
    resources:
            mem_mb= 32000,
            runtime= 1440
    conda:
            "envs/samtools.yaml"
    shell:
            "samtools sort -o {output} {input}"
```

Samtools sort rule

Samtools' sorting parameter description:
−o output = Output path

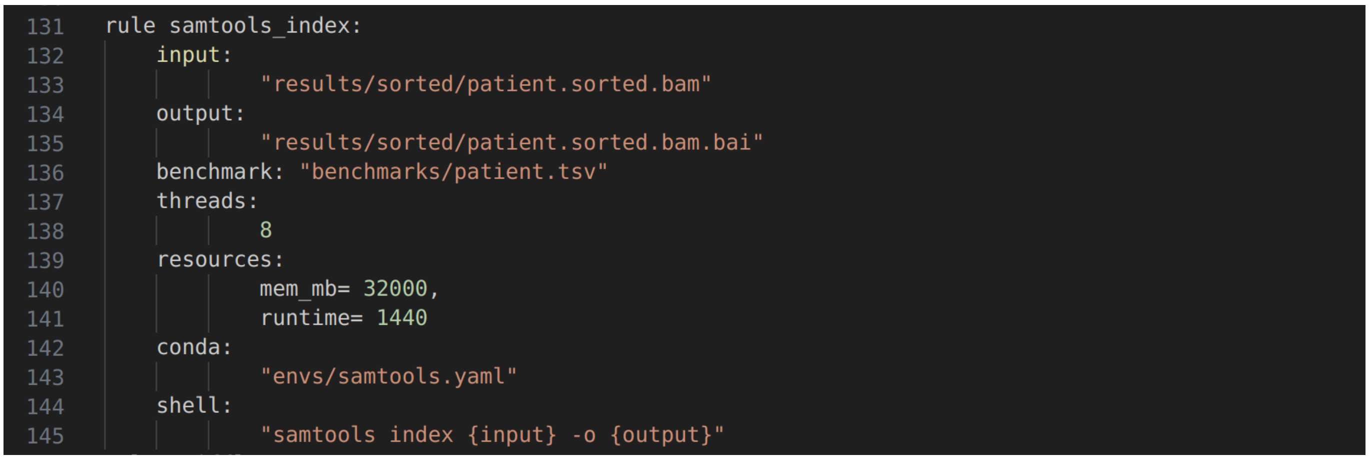


```
s         h:m:s   max_rss max_vms max_uss max_pss io_in    io_out   mean_load  cpu_time
1426.4534 0:23:46 858.97  955.04  854.31  855.02  31630.88 55097.63 97.68      1393.61
```

Samtools' sort used resources

Samtools index

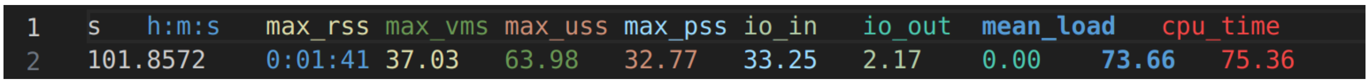
In this rule I perform the indexing, the last needed process in order to run variant calling. Structural variant callers in fact are based on algorithms that require an index to run; without it, that process would be significantly longer.



```
rule samtools_index:
    input:
            "results/sorted/patient.sorted.bam"
    output:
            "results/sorted/patient.sorted.bam.bai"
    benchmark: "benchmarks/patient.tsv"
    threads:
            8
    resources:
            mem_mb= 32000,
            runtime= 1440
    conda:
            "envs/samtools.yaml"
    shell:
            "samtools index {input} -o {output}"
```

Samtools index rule

Samtools' indexing parameter description:
−o output = Output path

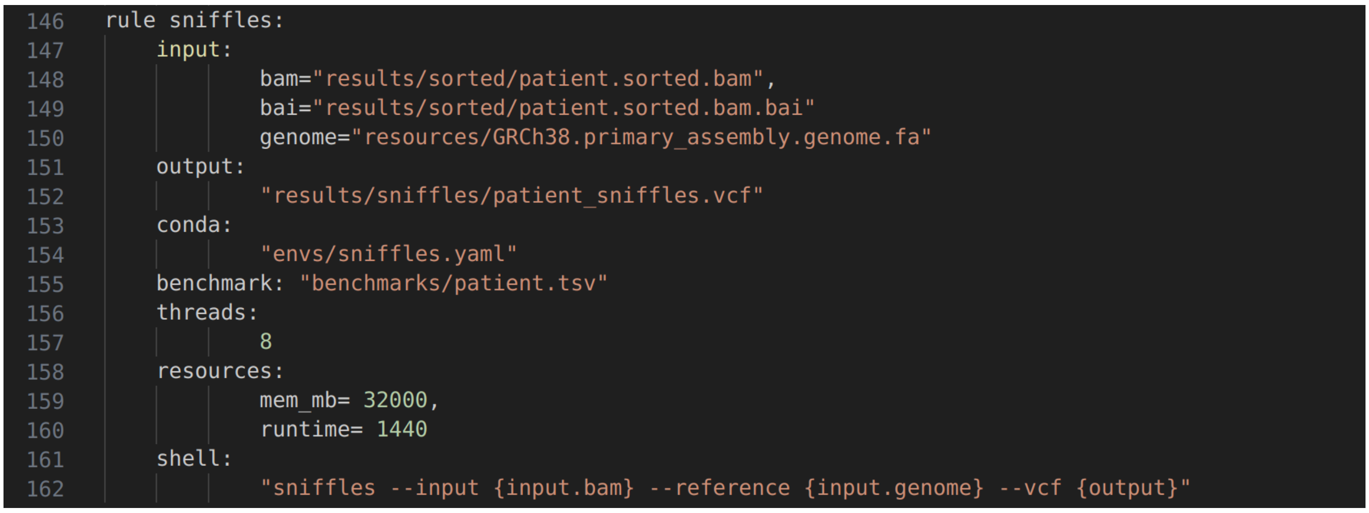


```
s        h:m:s   max_rss max_vms max_uss max_pss io_in io_out mean_load cpu_time
101.8572 0:01:41 37.03   63.98   32.77   33.25   2.17  0.00   73.66     75.36
```

Samtools' index used resources

3.2.12 Sniffles2

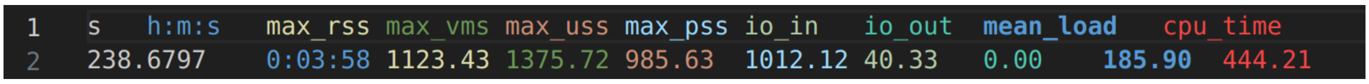
A fast structural variant caller for long-read sequencing, Sniffles2 accurately detects SVs on germline, somatic and population-level for PacBio and Oxford Nanopore read data. Through Sniffles2 I got a vcf file where all the structural variations are stored. Sniffles just require the alignment file to run.



```
rule sniffles:
    input:
            bam="results/sorted/patient.sorted.bam",
            bai="results/sorted/patient.sorted.bam.bai"
            genome="resources/GRCh38.primary_assembly.genome.fa"
    output:
            "results/sniffles/patient_sniffles.vcf"
    conda:
            "envs/sniffles.yaml"
    benchmark: "benchmarks/patient.tsv"
    threads:
            8
    resources:
            mem_mb= 32000,
            runtime= 1440
    shell:
            "sniffles --input {input.bam} --reference {input.genome} --vcf {output}"
```

Sniffles rule

Sniffles2 parameter description:
−−input input.bam = Input path
−−vcf output = Output path

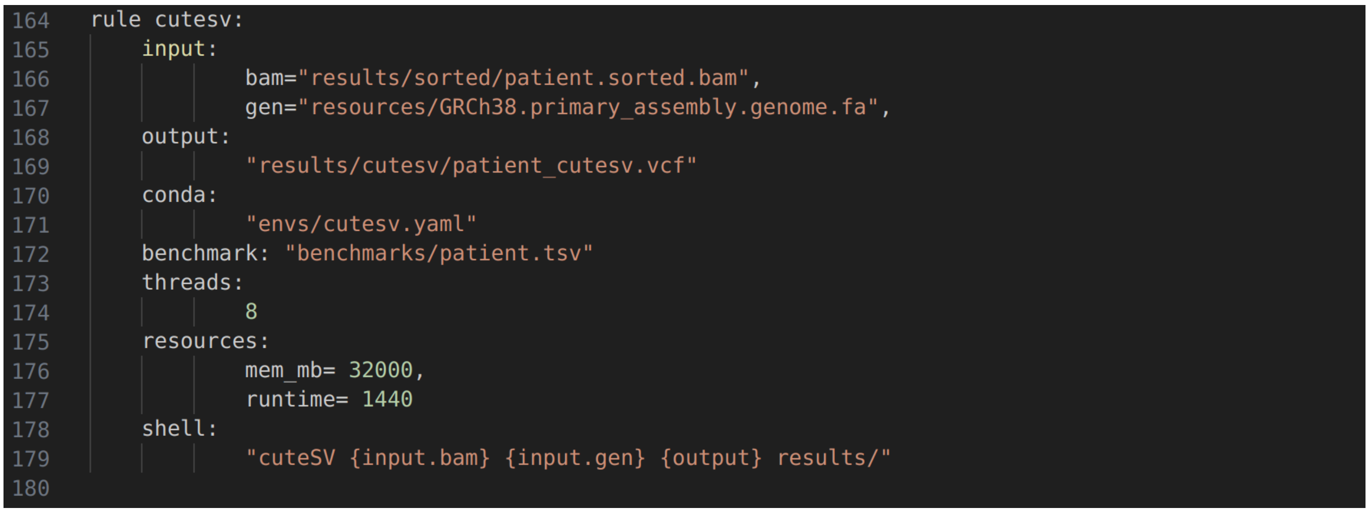


```
s        h:m:s    max_rss max_vms max_uss max_pss io_in   io_out  mean_load  cpu_time
238.6797 0:03:58 1123.43 1375.72 985.63  1012.12 40.33   0.00    185.90     444.21
```

Sniffles' used resources

3.2.13 CuteSV

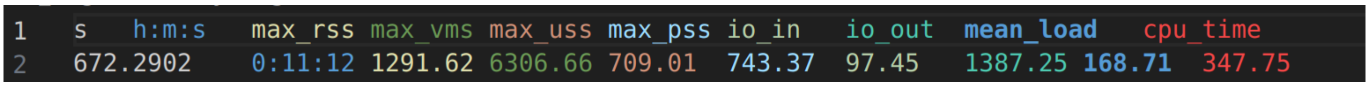
CuteSV was my second choice as S.V. caller. It is a really fast caller so I think it's interesting to compare it to Sniffles.



```
rule cutesv:
    input:
            bam="results/sorted/patient.sorted.bam",
            gen="resources/GRCh38.primary_assembly.genome.fa",
    output:
            "results/cutesv/patient_cutesv.vcf"
    conda:
            "envs/cutesv.yaml"
    benchmark: "benchmarks/patient.tsv"
    threads:
            8
    resources:
            mem_mb= 32000,
            runtime= 1440
    shell:
            "cuteSV {input.bam} {input.gen} {output} results/"

```

CuteSV rule

CuteSv requires as inputs a BAM file and the reference genome. So in the command line, I set the two inputs, the output directory and the working directory. In the latter folder, CuteSV creates some temporary output files while is working deleting them once done.

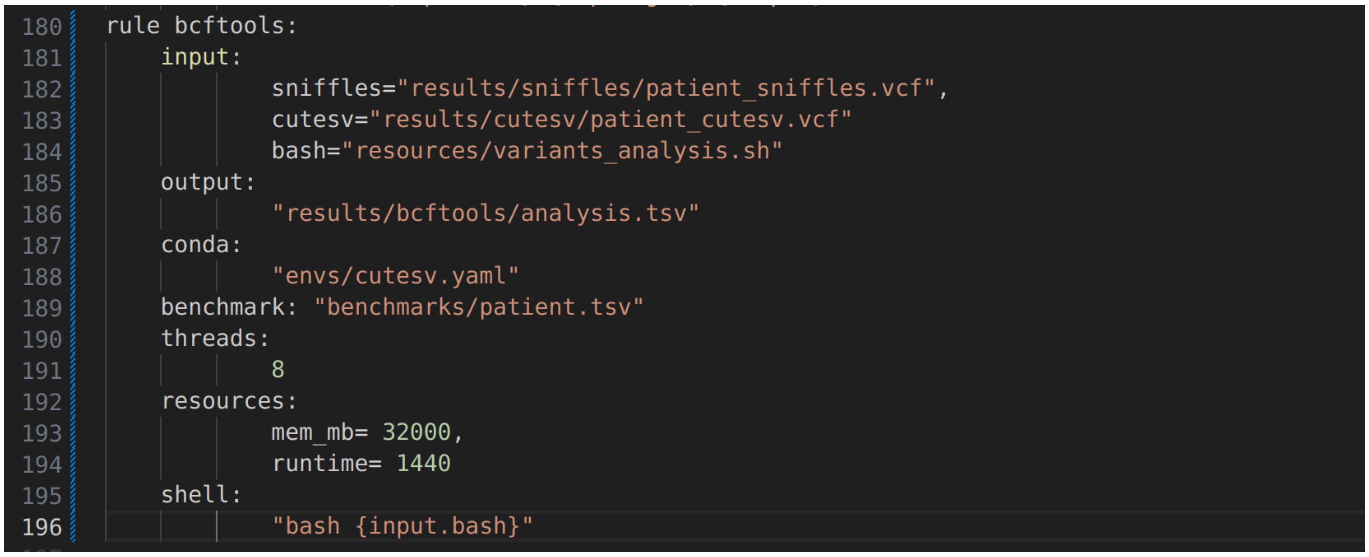


```
s        h:m:s    max_rss max_vms max_uss max_pss io_in   io_out  mean_load  cpu_time
672.2902 0:11:12 1291.62 6306.66 709.01  743.37  97.45   1387.25 168.71     347.75
```

CuteSV's used resources

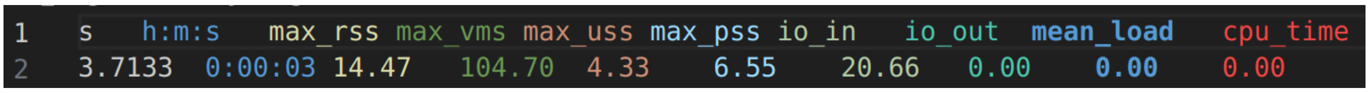
3.2.14 BCFtools

BCFtools is a program for variant calling and manipulating files in the Variant Call Format (VCF) and its binary counterpart BCF. [Danecek et al., 2021] I used BCFtools to extract information from my vcf files in order to understand the results of my work.



```
rule bcftools:
    input:
            sniffles="results/sniffles/patient_sniffles.vcf",
            cutesv="results/cutesv/patient_cutesv.vcf"
            bash="resources/variants_analysis.sh"
    output:
            "results/bcftools/analysis.tsv"
    conda:
            "envs/cutesv.yaml"
    benchmark: "benchmarks/patient.tsv"
    threads:
            8
    resources:
            mem_mb= 32000,
            runtime= 1440
    shell:
            "bash {input.bash}"
```

BCFtools rule

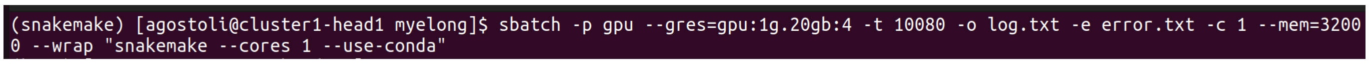


```
s      h:m:s   max_rss max_vms max_uss max_pss io_in  io_out mean_load  cpu_time
3.7133 0:00:03 14.47   104.70  4.33    6.55    20.66  0.00   0.00       0.00
```

BCFtools' used resources

3.2.15 Snakemake

Once coded the snakefile, I ran it in the cluster with the following command.



```
(snakemake) [agostoli@cluster1-head1 myelong]$ sbatch -p gpu --gres=gpu:1g.20gb:4 -t 10080 -o log.txt -e error.txt -c 1 --mem=3200
0 --wrap "snakemake --cores 1 --use-conda"
```

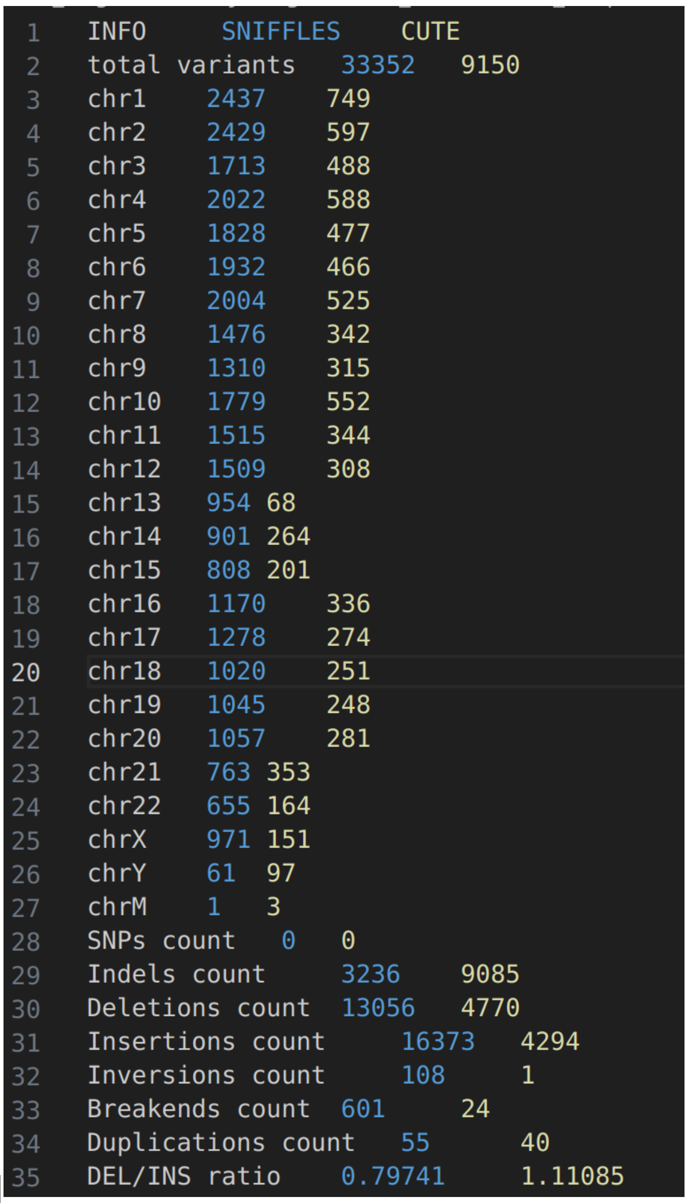
Cluster's parameter description:

−p gpu = Partition requested −−gres=gpu:1g.20gb:4 = Required generic resources −t 10000 = Time limit −o log.txt = File for batch script's standard output −e error.txt = File for batch script's standard error −c 1 = Number of cpus required per task −−mem=32000 = Minimum amount of real memory − − *wrap* "snakemake –cores 1 –use-conda" = Wrap command string in a sh script and submit

Snakemake's parameter description:

−−cores = Maximum number of cores requested from the cluster −−use−conda = If defined in the rule, run job in the specified conda environment

4 Results



```
INFO	SNIFFLES	CUTE
total variants	33352	9150
chr1	2437	749
chr2	2429	597
chr3	1713	488
chr4	2022	588
chr5	1828	477
chr6	1932	466
chr7	2004	525
chr8	1476	342
chr9	1310	315
chr10	1779	552
chr11	1515	344
chr12	1509	308
chr13	954	68
chr14	901	264
chr15	808	201
chr16	1170	336
chr17	1278	274
chr18	1020	251
chr19	1045	248
chr20	1057	281
chr21	763	353
chr22	655	164
chrX	971	151
chrY	61	97
chrM	1	3
SNPs count	0	0
Indels count	3236	9085
Deletions count	13056	4770
Insertions count	16373	4294
Inversions count	108	1
Breakends count	601	24
Duplications count	55	40
DEL/INS ratio	0.79741	1.11085
```

Figure 4.1: BCFtools output file

5 Discussion

With this project I was able to study structural variants in genomic DNAs from MM patients comparing different bioinformatics tools. The first thing I could see was that Sniffles detected a larger amount of structural variants than CuteSV, finding 33352 SV against 9150 of CuteSV. In both of them, the vast majority of SV were INDELS, almost equally divided into deletions and insertions. However, in Sniffles there are slightly more insertions than deletions, in fact in Sniffles there is a 0.79 deletions/insertions ratio. In contrast, there is a 1.11 deletions/insertions in CuteSV. I also found other SV like inversions, 108 in Sniffles and just 1 in CuteSv. Breakends were found as well: 601 in Sniffles and 24 in CuteSV. In the end, I discovered 55 duplications in Sniffles and 40 in CuteSV. In both the callers, the first two chromosomes are the ones that were more subjected to S.V.

6 Conclusions

With my project, i just scratched the surface of the detection of Structural Variants in Multiple Myeloma. In my results, i just extracted some basic information from my simple pipeline. This project could continue for example comparing other SV callers' performances, overlapping results. It would be possible to analyse just the SV detected by all the callers and ignoring the ones detected by just one of them. There are also more specific researches that could be done, such as looking for chromothripsis, chromoplexy and templated insertions which are significantly more complex variants detected in Multiple Myeloma studies. [Rustad et al., 2020]

Other analyses like searching for methylations, could be done in order to better understand the effects of Multiple Myeloma on DNA.

Bibliography

[Alt,] Altum. https://www.altumsequencing.com/services/multiple-myeloma/. Accessed: 2023-07-01.

[CNI,] Cnio. https://www.cnio.es/. Accessed: 2023-07-01.

[Con,] Conda. https://docs.conda.io/en/latest/. Accessed: 2023-07-01.

[Pan,] Pandas. https://pandas.pydata.org/. Accessed: 2023-07-01.

[Behjati and Tarpey, 2013] Behjati, S. and Tarpey, P. S. (2013). What is next generation sequencing? *Archives of Disease in Childhood - Education and Practice*, 98(6):236–238.

[Bird and Boyd, 2019] Bird, S. A. and Boyd, K. (2019). Multiple myeloma: an overview of management. *Palliative care and social practice*, 13:1178224219868235.

[Danecek et al., 2021] Danecek, P., Bonfield, J. K., Liddle, J., Marshall, J., Ohan, V., Pollard, M. O., Whitwham, A., Keane, T., McCarthy, S. A., Davies, R. M., and Li, H. (2021). Twelve years of SAMtools and BCFtools. *GigaScience*, 10(2). giab008.

[De Coster and Rademakers, 2023] De Coster, W. and Rademakers, R. (2023). Nanopack2: population-scale evaluation of long-read sequencing data. *Bioinformatics*, 39(5):btad311.

[De Vries et al., 2005] De Vries, B. B., Pfundt, R., Leisink, M., Koolen, D. A., Vissers, L. E., Janssen, I. M., Van Reijmersdal, S., Nillesen, W. M., Huys, E. H., De Leeuw, N., et al. (2005). Diagnostic genome profiling in mental retardation. *The American Journal of Human Genetics*, 77(4):606–616.

[Escaramís et al., 2015] Escaramís, G., Docampo, E., and Rabionet, R. (2015). A decade of structural variants: description, history and methods to detect structural variation. *Briefings in functional genomics*, 14(5):305–314.

[Hamdan and Ewing, 2022] Hamdan, A. and Ewing, A. (2022). Unravelling the tumour genome: The evolutionary and clinical impacts of structural variants in tumourigenesis. *The Journal of Pathology*, 257(4):479–493.

[Hupe et al., 2004] Hupe, P., Stransky, N., Thiery, J.-P., Radvanyi, F., and Barillot, E. (2004). Analysis of array cgh data: from signal ratio to gain and loss of dna regions. *Bioinformatics*, 20(18):3413–3422.

[Iafrate et al., 2004] Iafrate, A. J., Feuk, L., Rivera, M. N., Listewnik, M. L., Donahoe, P. K., Qi, Y., Scherer, S. W., and Lee, C. (2004). Detection of large-scale variation in the human genome. *Nature genetics*, 36(9):949–951.

[Jiang et al., 2020] Jiang, T., Liu, Y., Jiang, Y., Li, J., Gao, Y., Cui, Z., Liu, Y., Liu, B., and Wang, Y. (2020). Long-read-based human genomic structural variation detection with cutesv. *Genome biology*, 21(1):1–24.

[Li, 2018] Li, H. (2018). Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100.

[Mahmoud et al., 2019] Mahmoud, M., Gobet, N., Cruz-Dávalos, D. I., Mounier, N., Dessimoz, C., and Sedlazeck, F. J. (2019). Structural variant calling: the long and the short of it. *Genome biology*, 20(1):1–14.

[Marshall et al., 2008] Marshall, C. R., Noor, A., Vincent, J. B., Lionel, A. C., Feuk, L., Skaug, J., Shago, M., Moessner, R., Pinto, D., Ren, Y., et al. (2008). Structural variation of chromosomes in autism spectrum disorder. *The American Journal of Human Genetics*, 82(2):477–488.

[Mölder et al., 2021] Mölder, F., Jablonski, K. P., Letcher, B., Hall, M. B., Tomkins-Tinch, C. H., Sochat, V., Forster, J., Lee, S., Twardziok, S. O., Kanitz, A., et al. (2021). Sustainable data analysis with snakemake. *F1000Research*, 10.

[Nakagawa and Fujita, 2018] Nakagawa, H. and Fujita, M. (2018). Whole genome sequencing analysis for cancer genomics and precision medicine. *Cancer science*, 109(3):513–522.

[Rustad et al., 2020] Rustad, E. H., Yellapantula, V. D., Glodzik, D., Maclachlan, K. H., Diamond, B., Boyle, E. M., Ashby, C., Blaney, P., Gundem, G., Hultcrantz, M., et al. (2020). Revealing the impact of structural variants in multiple myeloma. *Blood cancer discovery*, 1(3):258–273.

[Sebat et al., 2007] Sebat, J., Lakshmi, B., Malhotra, D., Troge, J., Lese-Martin, C., Walsh, T., Yamrom, B., Yoon, S., Krasnitz, A., Kendall, J., et al. (2007). Strong association of de novo copy number mutations with autism. *Science*, 316(5823):445–449.

[Sebat et al., 2004] Sebat, J., Lakshmi, B., Troge, J., Alexander, J., Young, J., Lundin, P., Manér, S., Massa, H., Walker, M., Chi, M., et al. (2004). Large-scale copy number polymorphism in the human genome. *Science*, 305(5683):525–528.

[Sedlazeck et al., 2018] Sedlazeck, F. J., Rescheneder, P., Smolka, M., Fang, H., Nattestad, M., Von Haeseler, A., and Schatz, M. C. (2018). Accurate detection of complex structural variations using single-molecule sequencing. *Nature methods*, 15(6):461–468.

[Strozzi et al., 2019] Strozzi, F., Janssen, R., Wurmus, R., Crusoe, M. R., Githinji, G., Di Tommaso, P., Belhachemi, D., Möller, S., Smant, G., de Ligt, J., et al. (2019). Scalable workflows and reproducible data analysis for genomics. *Evolutionary Genomics: Statistical and Computational Methods*, pages 723–745.

[Tyson et al., 2005] Tyson, C., Harvard, C., Locker, R., Friedman, J., Langlois, S., Lewis, M., Van Allen, M., Somerville, M., Arbour, L., Clarke, L., et al. (2005). Submicroscopic deletions and duplications in individuals with intellectual disability detected by array-cgh. *American Journal of Medical Genetics Part A*, 139(3):173–185.

[Wick et al., 2019] Wick, R. R., Judd, L. M., and Holt, K. E. (2019). Performance of neural network basecalling tools for oxford nanopore sequencing. *Genome biology*, 20:1–10.

[Zhang, 2016] Zhang, H. (2016). Overview of sequence data formats. *Statistical genomics: Methods and protocols*, pages 3–17.