



Genome analysis

Cloud-native distributed genomic pileup operations

Marek Wiewióрка^{1,†}, Agnieszka Szurło^{1,†}, Paweł Stankiewicz² and Tomasz Gambin ^{1,*}

¹Institute of Computer Science, Warsaw University of Technology, Warsaw, Warsaw 00-661, Poland and ²Department of Molecular and Human Genetics, Baylor College of Medicine, Houston, TX 77030, USA

*To whom correspondence should be addressed.

[†]The authors wish it to be known that, in their opinion, the first two authors should be regarded as Joint First Authors.

Associate Editor: Peter Robinson

Received on August 27, 2022; revised on November 16, 2022; editorial decision on December 11, 2022; accepted on December 13, 2022

Abstract

Motivation: Pileup analysis is a building block of many bioinformatics pipelines, including variant calling and genotyping. This step tends to become a bottleneck of the entire assay since the straightforward pileup implementations involve processing of all base calls from all alignments sequentially. On the other hand, a distributed version of the algorithm faces the intrinsic challenge of splitting reads-oriented file formats into self-contained partitions to avoid costly data exchange between computational nodes.

Results: Here, we present a scalable, distributed and efficient implementation of a pileup algorithm that is suitable for deploying in cloud computing environments. In particular, we implemented: (i) our custom data-partitioning algorithm optimized to work with the alignment reads, (ii) a novel and unique approach to process alignment events from sequencing reads using the MD tags, (iii) the source code micro-optimizations for recurrent operations, and (iv) a modular structure of the algorithm. We have proven that our novel approach consistently and significantly outperforms other state-of-the-art distributed tools in terms of execution time (up to 6.5× faster) and memory usage (up to 2× less), resulting in a substantial cloud cost reduction. SeQuiLa is a cloud-native solution that can be easily deployed using any managed Kubernetes and Hadoop services available in public clouds, like Microsoft Azure Cloud, Google Cloud Platform, or Amazon Web Services. Together with the already implemented distributed range join and coverage calculations, our package provides end-users with a unified SQL interface for convenient analyses of population-scale genomic data in an interactive way.

Availability and implementation: <https://biodatageeks.github.io/sequila/>

Contact: tomasz.gambin@pw.edu.pl

1 Introduction

1.1 State-of-the-art

The sorted collection of the aligned sequencing reads can be transformed into a set of pileup records, also known as a coverage position summary. This format summarizes information about the base calls in all genomic positions from the reads aligned to a reference sequence, including total depth of coverage, non-reference (alternative) bases, and base qualities (see detailed definition in <http://www.htslib.org/doc/samtools-mpileup.html>). Pileup format was designed to provide the evidence of the single-nucleotide variants or the short insertions/deletions at given genomic positions. It is commonly used as an entry point to the well-established variant calling pipelines (Li, 2011) as well as to novel approaches to variant detection frameworks based on the neural networks (Luo *et al.*, 2020) or other methods, e.g. the binomial model, partial-order alignment, and de

Bruijn graph local assembly (Liu *et al.*, 2021) in fast variant calling. Coverage position summary is also used for identification of somatic mutations and copy number variation (Koboldt *et al.*, 2012).

Samtools suite (Li *et al.*, 2009) includes the mpileup tool, a gold standard for both data format and correctness of pileup calculations; however, it is a single-threaded program that does not provide the scalability feature (Sater *et al.*, 2020).

Research developments in the bioinformatics field emphasize a common need to use a technology that allows distributing long-lasting big data tasks into the multiple computing nodes or in the cloud computing infrastructure (Yuan and Wildish, 2020). In the recent Genome Analysis Toolkit (GATK, McKenna *et al.*, 2010) version, several programs (including pileup calculations) have been implemented in a distributed manner ready to be run on the Apache Spark cluster (Zaharia *et al.*, 2010). Other research studies confirm that big data programming paradigms can be successfully applied to

many genomic analyses (Capuccini et al., 2020; Guo et al., 2018; Wiewiórka et al., 2017, 2018) including variant calling (Ahmad et al., 2021). The analysis of the ever-increasing genomic datasets involves significant financial investments and administrative efforts to maintain secure and fault-tolerant storage solutions as well as fast and scalable processing units. To minimize those efforts, medical clinics and research centers consider migrating bioinformatics pipelines and custom analyses to private or public cloud infrastructure. The evolution towards cloud architecture is embraced by the widely used bioinformatics products both open-source (e.g. GATK) and commercial (e.g. DNA Nexus, Terra) (Koppad et al., 2021).

Although significant progress has been made, there are still areas in bioinformatics analyses that are not easily transferable to the distributed and cloud environments with traditionally used sets of tools.

1.2 Contribution

In the previous works, we proved that it was possible to implement very efficient and highly scalable tools, facilitating time-consuming common bioinformatics operations: interval joins [SeQuiLa-int algorithm (Wiewiórka et al., 2018), available in SeQuiLa package since version 0.3.0] and coverage calculations [SeQuiLa-cov algorithm (Wiewiórka et al., 2019), available in the SeQuiLa package since version 0.4.0]. In the next essential release of SeQuiLa (0.6.11) in 2021, we updated the code base to run on Apache Spark 3.1.2. We have also fixed the reported issues and improved the existing features; however, no major new functionalities have been introduced since version 0.4.0 (see detailed release history on <https://github.com/biodatageeks/sequila/tags>).

In the described current version (1.0.0), we have significantly extended the previously implemented functionality of the SeQuiLa package by introducing a novel distributed algorithm for summarizing reads using the Compact Idiosyncratic Gapped Alignment Report (CIGAR) strings and MD tags in a pileup format (SeQuiLa-pileup, Algorithm 2) (see Table 1). We also propose a custom data partitioning mechanism optimized to work with the alignment reads (Algorithm 1) as it significantly influences the performance of all subsequent steps by eliminating the need for data-exchange among partitions. At the same time, it enables single-pass over input data and lowers memory requirements since no intermediate data caching is required.

We have developed the algorithm in a modular way, enabling additional reduction of the execution time in two specific scenarios (when compared to regular SeQuiLa-pileup method), i.e. (i) pileup summary without information on base-qualities (denoted as SeQuiLa-pileup-cov-only) and (ii) depth of coverage information only (denoted as SeQuiLa-pileup-cov-only). Since the functionality of SeQuiLa-pileup-cov-only is the same as the one provided by the previously published SeQuiLa-cov tool (Wiewiórka et al., 2019) and the performance of the new algorithm (SeQuiLa-pileup-cov-only) is superior (see Section 3), we now recommend usage of SeQuiLa-pileup-cov-only for coverage calculations instead of SeQuiLa-cov while working with the current version of the SeQuiLa package (1.0.0).

Table 1. SeQuiLa package release history

Version (year)	Publication	Interval joins	Coverage	Pileup	Other features
0.3.0 (2018)	Wiewiórka et al. (2018)	<i>int</i>	—	—	—
0.4.0 (2019)	Wiewiórka et al. (2019)	<i>int</i>	<i>cov</i>	—	—
0.6.11 ^b (2021)	—	<i>int</i>	<i>cov</i>	—	—
1.0.0 (2022)	—	<i>int</i>	<i>pileup-cov-only</i> ^a	<i>pileup</i> ^a	<i>reads-aware partitioning</i> ^a , <i>cloud recipes</i> ^a

Note: *int*, interval joins (SeQuiLa-int); *cov*, coverage calculations (SeQuiLa-cov); *pileup-cov-only*, coverage calculations using simplified pileup algorithm (SeQuiLa-pileup-cov-only); *pileup*, pileup calculations (SeQuiLa-pileup).

^aNovel tools and features described in this manuscript.

^bTechnical update (Apache Spark update and fixed reported issues).

Besides the custom partitioner and pileup algorithm, the new version of the SeQuiLa package provides Terraform modules, Docker images, and code examples that facilitate straightforward deployment in the public clouds infrastructure.

2 Materials and methods

2.1 Rationale

The foundations for the distributed pileup algorithm are based on three key observations.

Firstly, the majority of bases in the aligned sequencing reads are concordant with the reference sequence. Therefore, we designed our algorithm to use both CIGAR strings, representing spliced alignment operations and MD tags, encoding mismatched and deleted reference bases, as defined in <https://samtools.github.io/hts-specs/SAMv1.pdf> and <https://samtools.github.io/hts-specs/SAMtags.pdf> accordingly. The use of the above mentioned strings in conjunction allows to handle deletions, insertions, and substitutions without decoding and parsing the entire read sequence and base qualities. To the best of our knowledge, it is the only algorithm that takes advantage of this information for pileup construction.

Secondly, our pileup computation is divided into four units of work: (i) coverage computation, (ii) identification of non-reference base calls, (iii) collection of base qualities and (iv) output projection. This decomposition allows us to reduce computational complexity by skipping certain steps that are not required.

Finally, the most limiting factor of performance and scalability for any distributed processing is the data exchange among the worker nodes that always requires costly data serialization as well as network transfer. Therefore, we propose a new data partitions coalescing mechanism, which guarantees proper handling of reads overlapping more than one partition without the need of data shuffling. In addition, we use BAM indexes for efficient partition boundaries adjustment and thus significantly reducing input/output (IO) operations.

2.2 Algorithm

2.2.1 Defining partitions

Consider an input sorted collection of the aligned sequencing reads R divided into n partitions by the underlying file system (Fig. 1A). The set of all partitions constitutes an immutable collection of data, i.e. resilient distributed dataset (RDD) which is the main logical unit of data in the Apache Spark framework.

For each partition, we calculate two values: lower and upper bounds that create self-contained virtual read partitions ($V1 = lb1 - ub1$, $V2 = lb2 - ub2$, etc, from Fig. 1C, Algorithm 1). For clarity, in pseudo-code we assume that all reads are aligned to a single chromosome.

This information is further required for changing the default Apache Spark partitioning schema ($P1, P2, \dots, PN$ —see Fig. 1A), creating new coalesced partitions ($C1, C2, \dots, CN$ —see Fig. 1D). Within each coalesced partition, the algorithm processes only the reads overlapping with the corresponding virtual partition (see

Algorithm 1 Reads-aware partitioning: Calculating lower(lb) and upper(ub) bounds for self-contained read partitions

Require: P , RDD (Resilient Distributed Dataset) containing all reads' partitions

```

1. for  $i \in (0, \text{length}(P) - 1)$  do
2.    $r(i) \leftarrow p(i)(0)$  get the first read in  $i$  partition
3.    $lb(i) \leftarrow r(i)_{start}$  get the lower bound of  $i$  partition
4. end for
5. for  $j \in (0, \text{length}(P) - 1)$  do construct interval tree of all
   reads overlapping any of  $lb(j)_{start}$ 
6.    $it \leftarrow \text{IntervalTree}(p(i).getReadsOverlapping(lb(j)_{start}))$ 
7. end for
8. for  $i \in (0, \text{length}(P) - 1)$  do
9.   if  $i \neq (\text{length}(P) - 1)$  then
10.     $ub(i)_{pos} \leftarrow \max(it.overlappers(lb(i+1)_{pos}))$ 
11.   else
12.     $ub(i)_{pos} \leftarrow \text{Int.maxValue}$ 
13.   end if
14.   if  $i > 0$  then
15.     $lb(i) \leftarrow ub(i-1)$ 
16.   end if
17. end for
18. return ( $lb, ub$ )

```

Fig. 1E). Note that our approach is very lightweight—it dynamically calculates boundaries for virtual partitions, which can be considered as view upon original Spark partitions. Unlike GATK which divides input data into fixed length multikilobase-size pieces called *shards*, we do not introduce any other level of data splitting and push all computations down to the partition level.

2.2.2 Calculating coverage and alternative alleles

For each virtual partition, the program generates an aggregate object holding: (i) an array of alignment events (i.e. start and end of alignment) which is gathered for the event-based coverage calculations, (ii) a map of alternative bases count calculated using read MD tag and (iii) an interval tree structure of succinct read representation—*ReadSummary* (i.e. start, end, CIGAR derived configuration) used for further base qualities calculations. The program calculates base qualities only for the positions where at least one alternative base is present (Algorithm 2).

2.2.3 Merging and rendering the results

Once all reads in each virtual partition are analyzed, the program calculates the set of final pileup records. Depending on the configuration, our modular pileup algorithm can generate different outputs: full pileup (SeQuiLa-pileup), pileup without base qualities (SeQuiLa-pileup-no-qual) or depth of coverage only (SeQuiLa-pileup-cov-only).

2.3 Technical design

Our algorithm is implemented as a plugin to Apache Spark Catalyst optimizer (Armbrust *et al.*, 2015). We used its three extension points: (i) SQL Analyzer—to register new table-valued functions, (ii) Planner—to add our optimized execution strategies for pileup calculations and (iii) Logical Optimizer—to detect *CreateDataSourceTableAsSelectCommand* and *InsertIntoHadoopFsRelationCommand* actions and apply optimizations for direct vectorized writes into the Optimized Row Columnar (ORC) files (Fig. 2).

Algorithm 2 SeQuiLa-pileup

Require: ref : Reference sequence

$conf$: Configuration

$PartitionSet$, RDD (Resilient Distributed Dataset) containing all reads' partitions

```

procedure CALCULATEPILEUP( $ref, conf$ )
2:   for  $p \in PartitionSet$  do
        $aggregates := assembleAggregates(p, conf)$ 
4:    $generatePileupRecords(aggregates, ref, conf, p.lb, p.ub)$ 
   end for
6: end procedure
procedure ASSEMBLEAGGREGATES( $partition, conf$ )
8:    $agg := initAggregateForContig$ 
   for  $read \in partition$  do
10:     $agg.events := calculateCoverageEvents(read)$ 
    if  $conf.includeAlts$  then
12:       $agg.alts += calculateAlts(agg, read, MDTag)$ 
       $agg.treeCache += createReadSummary(agg, read)$ 
14:    end if
   end for
16: end procedure
procedure GENERATERECORDS( $aggregates, ref, conf, lb, ub$ )
18:   for  $a \in aggregates$  do
       for  $pos \in 0..ub$  do
20:          $sum := cumulativeSum(a.events, pos)$ 
         if  $pos \geq lb$  and  $coverageChange$  or  $hasAlt$  then
22:           if  $conf.includeQuals$  then
                $quals := calculateQuals(a.treeCache, pos)$ 
24:           end if
                $createRow(sum, ref, a.alts, quals)$ 
26:         end if
       end for
28:   end for
end procedure

```

We designed the relational model to represent alignments and pileup function results as proposed in Sun *et al.* (2018) and Smith *et al.* (2021). Our package provides both SQL (Structured Query Language) and Dataframe programming interfaces for the Scala and Python (<https://github.com/biodatageeks/pysequila>) languages.

For reading Binary Alignment Map (BAM) and Compressed Reference-oriented Alignment Map (CRAM) files our solution can use Hadoop-BAM (Niemenmaa *et al.*, 2012) or disq libraries as configured by the end-user. For better support of the CRAM files that have been recently added to the HTSJDK library, we extended the Hadoop-BAM project (<https://github.com/biodatageeks/Hadoop-BAM>). Also, minor changes required for serialization of genomic intervals parameters were added to the disq (<https://github.com/mwiewior/disq>) library. For saving output we support not only ORC but also Parquet file format (Ivanov and Pergolesi, 2020). In our code, we re-used partition coalescing mechanism as implemented in the GATK.

2.4 Essential optimizations

Our main goal was to deliver a fast, distributed and scalable implementation of the pileup algorithm. In addition to the already presented novel algorithm, we highlight other essential implementation decisions that improve overall software performance. They can be grouped into three main categories: (i) optimization of distributed

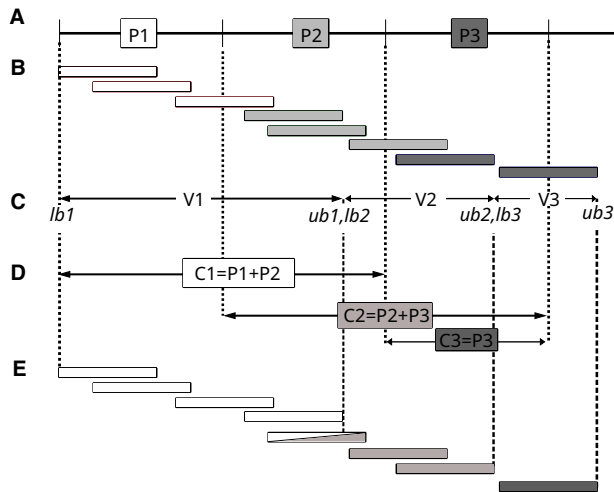


Fig. 1. Reads-aware partitioning algorithm: original distributed partitions (A); read assignment (color coded) to original partitions according to alignment starting position (B); virtual partitions and their boundaries calculated by Algorithm 1 (C); coalesced partitions (D); and read assignment (color coded) to coalesced partitions and corresponding virtual partitions (E). Note that some of the reads will be processed in more than one coalesced partition. This approach produces on average equally sized virtual partitions (no data skewness) except for the first one and last one that are a bit larger and smaller than the rest, respectively

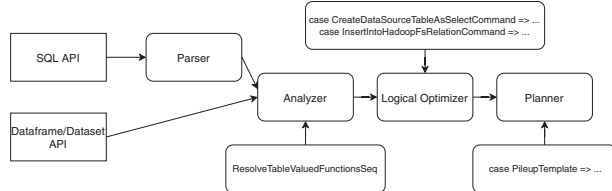


Fig. 2. SeQuiLa extensions to Apache Spark Catalyst optimizer

processing, (ii) Scala source code micro-optimizations, and (iii) output vectorization and fine-tuning.

2.4.1 Optimization of distributed processing

In the straightforward approach where the default partitioning is used, pileup implementation in Apache Spark would be split into two stages with a data shuffle step in between. This would require either explicit caching of the intermediate results from the first one or at least partial recomputing of the evicted partitions to get the final results. Implementation of a custom partitioning mechanism (Algorithm 1) to appropriately split data and determine the boundaries for each split was essential to achieve a single-pass solution without any extra data exchange between the executors or caching intermediary results.

2.4.2 Source code micro-optimizations

We have observed an apparent speedup when using an interval tree to store a short representation of reads (*ReadSummary*) since data retrieval from this structure is performed frequently with interval conditions. After analyzing the profiling results in a form of flame graphs obtained with *async-profiler* (Nisbet et al., 2019), we identified the most time-consuming and frequently invoked methods, such as calculation of the relative position in a read for a given genomic coordinate, and re-implemented them in the state-aware manner, thus eliminating traversing collection on each call. Similarly, we substituted computationally expensive CIGAR parsing and interpretation with fast lookups to lazily evaluated custom objects with derived cigar configuration with quick checks for existence of clip (and its length) or deletion, as well as deletion and insertion positions.

2.4.3 Output and auxiliary optimizations

The default output generation mechanism, which accounted for around 30% of the total processing time of our algorithm turned out to be another bottleneck. Therefore, we have implemented two novel approaches for optimizing output rendering.

Firstly, we have developed a custom direct pileup record projection to Apache Spark's internal binary row representation applying several micro-optimizations, e.g. casting reference bases and contig names to bytes and caching them in map to avoid repeating this task for each record. This mechanism can be used for further processing pileup rows within Spark-SQL engine as well as for persisting the results in any supported file format.

The second optimization is intended for improving performance of saving the results in ORC file format only. Inspired by the idea of *direct-path load* introduced in the relational database management systems, in particular in Oracle database (Heller, 2019), we implemented a mechanism that enables bypassing Spark's internal data representation and provides the support for vectorized row batches (as proposed in Shen et al., 2021) that are used for producing ORC output.

Other auxiliary optimizations including external dependencies configuration and environment setup were evaluated, and their impact on the overall performance is described in Section 3.

2.5 Cloud readiness

The increasing availability of cloud computing services for research is gradually changing the way scientific applications are developed, deployed and run (Vaillancourt et al., 2020). To ensure portability and reproducibility of SeQuiLa-based data processing, we followed the Infrastructure as Code (Guerrero et al., 2019) and DevOps principles for setting up the computing resources that can be used for both private and public clouds deployments. Hence, we have used technologies like Terraform (for cloud infrastructure provisioning, Modi, 2021), Helm (for deploying applications on Kubernetes clusters, Shah and Dubaria, 2019), and Docker (for application code packaging and shipment, Boettiger, 2015). SeQuiLa has been successfully deployed to both popular managed Hadoop services like Google Dataproc (utilized also in Krissaane et al., 2020) and managed Kubernetes services like Google Kubernetes Engine (GKE), Azure Kubernetes Service, or Amazon Elastic Kubernetes Service. Figure 3 presents an exemplary setup on GKE using the spark-on-k8s-operator and SeQuiLa application defined as a Kubernetes Custom Resource Definition. This architecture was suggested in Castro et al. (2019) as a preferred Apache Spark deployment scenario for scaling data analytics workloads and enabling efficient, on-demand utilization of resources in the cloud infrastructure. More detailed information on setup and corresponding Terraform modules can be found in the dedicated GitHub repository (<https://github.com/biodatageeks/sequila-cloud-recipes>).

2.6 Features

Table 2 summarizes the features of the SeQuiLa package and compares them with state-of-the-art software, including samtools and GATK.

SeQuiLa-pileup operates on sorted aligned sequencing reads both in BAM and CRAM format. The fast pileup algorithm requires reads to have MD Tag attribute which can be determined during the alignment process or calculated and added to BAM files independently after alignment is completed. MD tag is described in <https://samtools.github.io/hts-specs/SAMtags.pdf> as a string encoding the mismatched and deleted reference bases, used in conjunction with the CIGAR and SEQ fields to reconstruct the bases of the reference sequence interval to which the alignment has been mapped. This can enable variant calling without requiring access to the entire original reference. Input files can be read either from the local file system, distributed file system or object storage using a custom data source which allows representing input reads as relational data. The dataset used for calculating the pileup can be restricted according to the

user-provided parameters including reads bit flag and mapping quality.

Samtools and GATK produce verbose output for every coordinate. On the contrary, our software implements lossless block compression of adjacent genomic positions which results in output an order of magnitude smaller. SeQuiLa-pileup result includes the genomic coordinates, reference bases, depth of coverage, the ratio of reference to non-reference bases, alternative bases (strand-aware) with occurrences counts and optionally the base qualities for the positions where at least one non-reference base is present. The output is stored in the popular big data-ready file formats such as ORC or Parquet making it easy to run further analyses, e.g. in Apache Spark or tools like Trino (Sethi et al., 2019).

The SeQuiLa package is also distributed as a Python module (<https://github.com/biodatageeks/pysequila>) and can be used on local resources or cloud infrastructure. It can be easily integrated with widespread open-source notebook-based environments for data analysis including Google Colab and Jupyter.

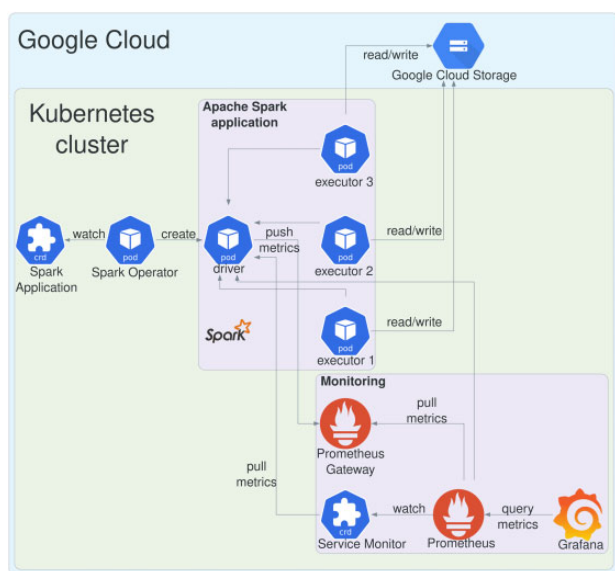


Fig. 3. SeQuiLa deployment on GKE with spark-on-k8s-operator with Kubernetes Custom Resource Definition, Prometheus for runtime metrics collection and Grafana as observability platform

3 Results

3.1 Datasets

We have used publicly available Exome Sequencing (ES) and Whole Genome Sequencing (WGS) datasets. We performed quality assurance tests on both short reads (sample NA12878) and long reads (guppy), represented in BAM and CRAM formats that were aligned to human reference genome GRCh38 with MD tags included.

3.2 Investigated solutions

Table 2 summarizes the functionalities of tools included in our comparison. Among the solutions included in the benchmark, only Spark-based GATK and SeQuiLa offer both multi-threaded and distributed versions of the depth coverage and pileup algorithms. For ADAM and SeQuiLa we used Apache Spark 3.1.2 runtime, in the case of GATK that does not provide support for Spark 3.x, we used Apache Spark 2.4.3. Megadepth, mosdepth and samtools 1.14 are multi-threaded applications but only the parts of their algorithms responsible for the IO operations (BZGF block compression/decompression) are parallelized—the remaining stages of their algorithm are sequential.

We have also included the previous version of SeQuiLa software (0.6.11) to assess the improvement of our single-pass and cache-less SeQuiLa-pileup-cov-only algorithm over the previously published SeQuila-cov. Several tools require additional input, i.e. genomic intervals in case of GATK’s coverage and PaCBAM’s (Valentini et al., 2019) pileup or the list of genomic positions in case of aseq’s (Romanel et al., 2015) pileup that restricts the processed data and affects algorithm’s computational complexity therefore the aforementioned solutions were not included in the final benchmark.

3.3 Testing environment

3.3.1 Single machine

Table 3 presents key information regarding the hardware and operating system configuration of the machine used for benchmark purposes. No hardware or software virtualization was used.

3.3.2 Hadoop cluster

Hadoop cluster (HDP 3.1.4) consists of 6 master and 34 worker nodes, 680 (1360 logical) cores, 700 TB of Hadoop File System (HDFS) disks, 6.8 TB of RAM for Yet Another Resource Negotiator (YARN) node pool and a 100 Gbits interconnect network. Master node specification was the same as in the single-node benchmark, in the case of workers the only difference was in disks configuration—

Table 2. Investigated solutions

Tool	Coverage	Pileup	MQF	SA	MT	DIST
samtools 1.9 (Li et al., 2009)	+	+	+	+	-	-
samtools 1.14 (Danecek et al., 2021)	+	+	+	+	+	+
GATK 4.2.3.0 (McKenna et al., 2010)	+	+	+	+	+	+
GATK-Spark 4.2.3.0	+	+	-	-	+	+
ADAM 0.36.0 (Massie et al., 2013)	+	-	-	-	+	+
megadepth 1.1.1 (Wilks et al., 2021)	+	-	-	-	+	-
mosdepth 0.3.2 (Pedersen and Quinlan, 2018)	+	-	-	-	+	-
sambamba 0.8.1 (Tarasov et al., 2015)	+	-	-	-	-	-
SeQuiLa 0.6.11 (Wiewiórka et al., 2019)	cov	-	-	-	+	+
SeQuiLa 1.0.0	pileup-cov-only	pileup	+	+	+	+

MQF, Mapping Quality Filter; SA, strand-awareness; MT, multi-threaded; DIST, distributed.

Table 3. Technical specification—single node.

Processor	Base freq (GHz)	CPUs	Total cores (logical)	Memory (GB)	Operating system	Disk
Intel(R) Xeon(R) E5-2618L v4	2.20	2	20 (40)	256	RHEL 7.8 (Maipo)	3TB (RAID1)

each node has additional 12 disks in Just a Bunch of Disks setup for HDFS storage.

3.4 Performance testing scenarios and configuration

We have arranged four testing scenarios: (i) the pileup function performance on the local machine and (ii) its scalability characteristics on the Hadoop Cluster, (iii) the depth of coverage function performance on the local machine and (iv) its scalability on the Hadoop Cluster. All tools from Table 2 were included in the presented benchmarks. ES and WGS alignment datasets in the BAM format have been used as inputs. In the case of tools (ADAM, GATK and SeQuiLa) running on top of Java Virtual Machine (JVM) we used three distributions of Java Development Kit (JDK)—for a single node, we used GraalVM CE JDK8 for GATK (it does not support JDK11 yet) and GraalVM CE JDK11 for ADAM and SeQuiLa for running tests on the Hadoop cluster OpenJDK8 was used for all solutions. For tests using `disq` library, an additional BAM index was created.

3.5 Results pileup

In the pileup benchmarks (Fig. 4), SeQuiLa-pileup proved to be the fastest tool outperforming samtools in the single thread scenarios by $1.25\times - 1.4\times$ and GATK (both Spark, and non-Spark based) $\approx 3.9 - 6.5\times$ GATK (both Spark, and non-Spark based). In the case of Hadoop cluster benchmarks SeQuiLa-pileup again proved to be faster by $\approx 2.8 - 5.3\times$ than GATK that also required twice as much memory (8 instead of 4GB) per Spark executor to be able to complete the computations. It is worth noting that we were unable to run GATK with 10 or fewer Spark executors (10 cores) facing errors related to too many opened files (even after increasing Linux

`nofile` limit to more than 1 million that is more than the recommended value for Hadoop clusters). We have verified that the algorithm's modularity is gainful when the user does not need to obtain the full pileup summary statistics. In particular, if reporting of base qualities is not required, SeQuiLa-pileup-no-qual that improves the performance by $\approx 35\%$ (compare SeQuiLa-pileup and SeQuiLa-pileup-no-qual in Fig. 4A) can be used. The correctness of the algorithm output was ensured by its rigorous comparison to the results of Samtools mpileup (v 1.14).

3.5.1 Java virtual machine optimizations

For development and local testing purposes, we have chosen GraalVM which uses an optimized compiler, generating high-performance code, and therefore noticeably accelerates the execution of the JVM-based applications (Sipek et al., 2020). Additionally, on the source code level, we have applied inlining annotations for frequently called concise methods which are further handled by the Scala compiler, thus avoiding the overhead of method invocation. In our diagnostic tests, we have confirmed that GraalVM choice results in 15% speedup while in-lining improved the timing by another 3% (Fig. 5).

3.5.2 Input-output optimizations

When performing direct vectorized writes into ORC files we have saved 18% of the computing time. We also take advantage of Intel's Genomics Kernel Library (GKL) providing high-performance operations of decompressing BAM file records. Our benchmarking confirms that the proper use of GKL's methods results in a 12% decrease of compute time (Fig. 5).

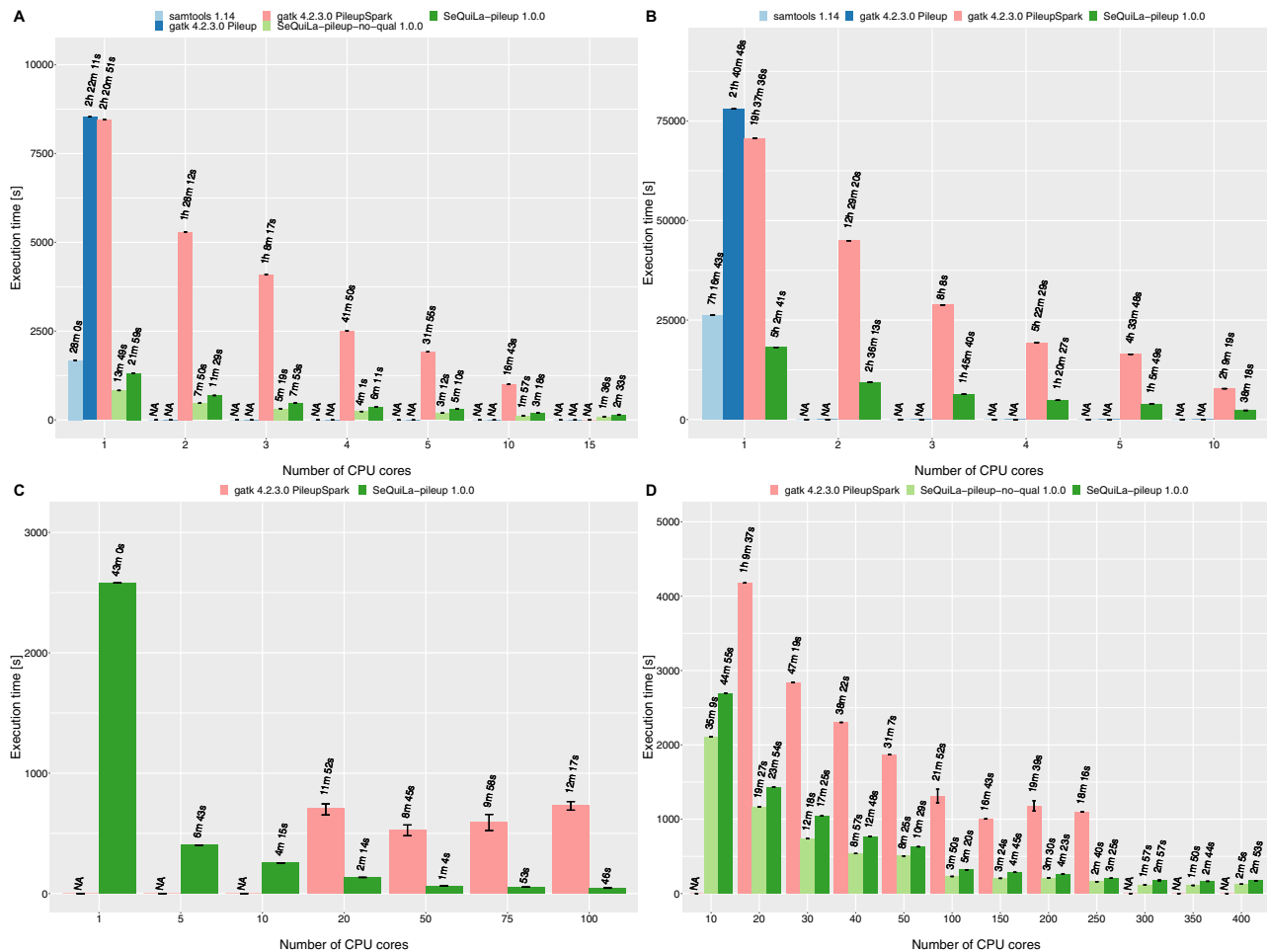


Fig. 4. Pileup summary function comparison. Tests were performed on a single node for ES (A), WGS (B), and on the Hadoop cluster for ES (C), WGS (D)

3.6 Results depth of coverage

In the case of both ES and WGS datasets (Fig. 6), megadePTH proved to be the fastest tool in a single-machine single thread setup outperforming the second one, SeQuiLa-pileup-cov-only by 1.3–1.6×. The gap between them decreases steadily with an increasing

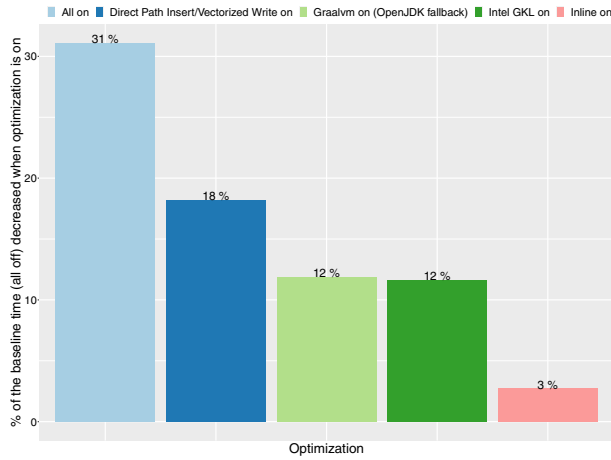


Fig. 5. Impact of various optimizations techniques on overall performance (percentage of time reduction) as compared with baseline (all optimizations off) for pileup computation. First bar shows the performance gain when all optimizations are on

number of threads. While processing ES and WGS data SeQuiLa-pileup-cov-only becomes the fastest tool when 5 and 10 threads are used, respectively. It is worth emphasizing that for the following tools: megadePTH, mosdePTH, and samtools, we observed very similar performance characteristics—in contrast to SeQuiLa-pileup-cov-only they do not scale up beyond 5–10 threads at all. These results confirm the fact that these tools only implement the parallel read and blocks decompression operations and the main part of their algorithms does not take advantage of multiple cores. For ADAM, we only measured the single-threaded performance that proved to be substantially worse than the remainder of the best performing tools (≈40x). Last but not least, we confirmed that the current version of the coverage calculations algorithm implemented in SeQuiLa 1.0.0 package (SeQuiLa-pileup-cov-only) is approximately 2× faster than our previous version of the coverage algorithm (SeQuiLa-cov) described in (Wiewiórka et al., 2019) [compare SeQuiLa-pileup-cov-only (1.0.0) and SeQuiLa-cov (0.6.11) on Fig. 6A].

In the case of WGS on the Hadoop cluster (Fig. 6), we benchmarked tools allocating from 10 to 200 cores. SeQuiLa-pileup-cov-only outperformed ADAM on average by more than an order of magnitude (11 – 16x). We used the same memory (8GB—driver, 4GB executor) and Central Processing Unit configurations (1 core) for both Spark processes to ensure comparability of the results between solutions. Also Spark dynamic allocation mechanism has been explicitly disabled. In the case of ADAM tests, we observed random fails of Spark tasks (or even the whole stages) due to network timeouts.

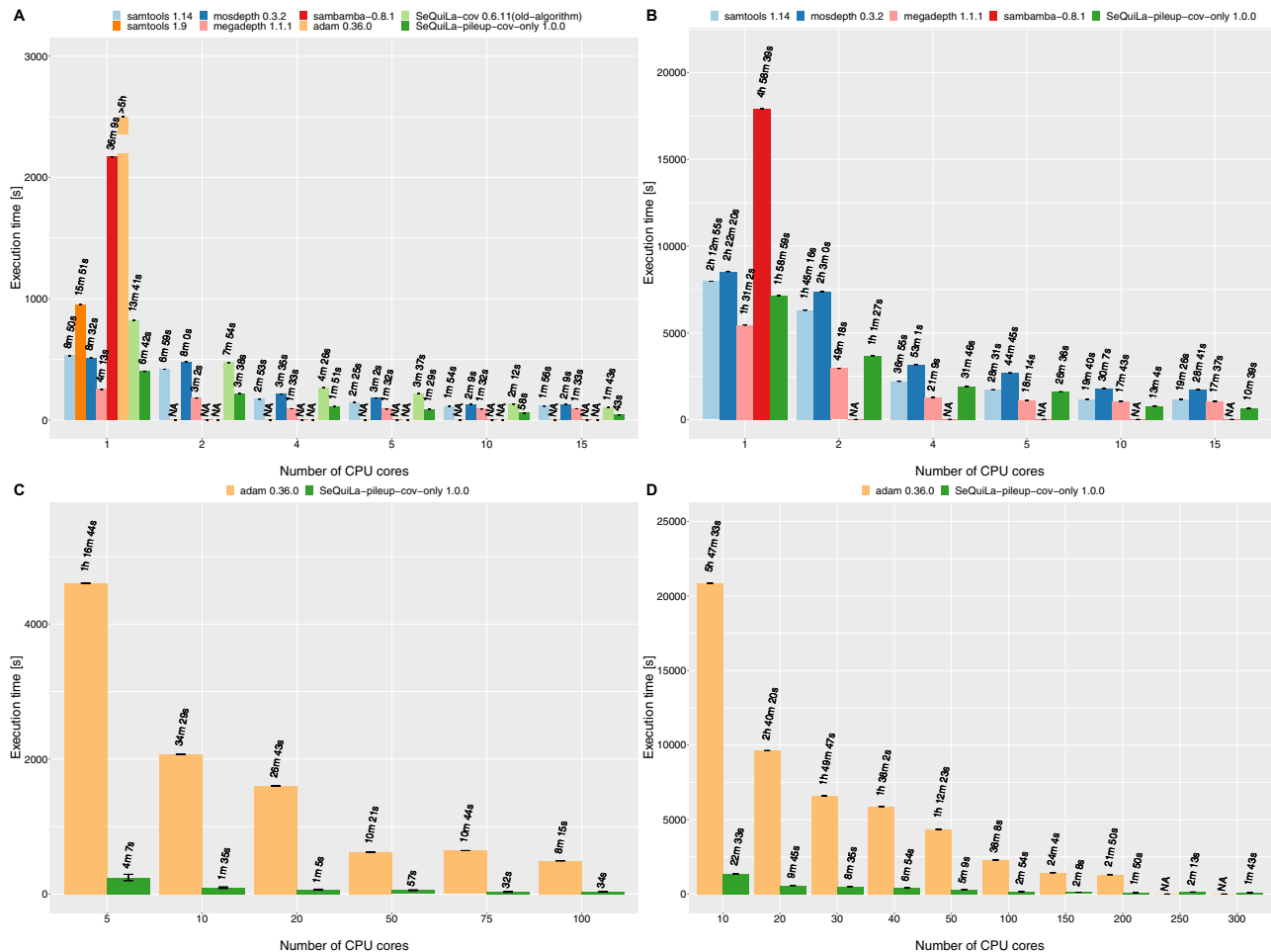


Fig. 6. Depth of coverage function comparison. Tests were performed on a single node for ES (A), WGS (B), and on the Hadoop cluster for ES (C), WGS (D). SeQuiLa-pileup designates the execution time of the full pileup calculations; SeQuiLa-pileup-no-qual indicates the execution time of the simplified pileup calculations in which base qualities are not computed

Discussion

Our pileup method is designed for compatibility with modern distributed computing systems originating from the Hadoop ecosystem mostly implemented using JVM languages such as Java or Scala. This approach incurs additional overheads and causes inefficiencies in the single-node deployments that justify why it does not significantly outperform samtools (a tool written in C language compiling to the native code) in a single thread comparisons. Our pileup algorithm performs especially well on the alignment files with the high-quality short reads when MD tags contain a relatively low number of mismatch/deleted bases. Calculating the complete pileup summaries from the long reads with a large number of mismatches is more challenging for our approach and requires additional modifications that we plan to introduce in its future versions. This limitation does not apply to calculations of the depth of coverage. It also favors BAM over CRAM alignment file formats (data not shown). This is because both the alignment file index scans (random access) as well the sequential reads are much slower ($\approx 3 - 4x$) in the case of CRAM when compared to BAM file format (our results confirm the findings presented in Supplementary Material) (Bonfield et al., 2019). Finally, saving results in the distributed processing can be substantially reduced with adding support for direct, vectorized writes (currently available in the local mode) that is on our project roadmap as well.

Since the complexity of the cloud-native distributed computing systems have been acknowledged in many studies, including Vaillancourt et al. (2020), we have also prepared ready-to-use cloud deployment examples that can help users to start using SeQuila in public cloud environments.

4 Conclusions

We present a new module that extends and optimizes our SeQuila Apache Spark library. This component introduces a new algorithm for fast, scalable, and fully distributed computation of pileup summary from the alignment files (BAM, CRAM). Our solution combines a distributed computing engine based on the extended Apache Spark Catalyst query optimizer with the SQL interface for handling large-scale processing and analyzing next-generation sequencing datasets in a consistent tabular form. This approach will help to facilitate the adoption of scalable solutions among users that are neither proficient in distributed computing nor in cloud infrastructures as envisioned in Lawlor and Sleator (2020).

Funding

This work was supported by the (POB Cybersecurity and data analysis) of Warsaw University of Technology within the Excellence Initiative: Research University (IDUB) program. Grant's Principal Investigator: A.S.. The funding body did not play a role in the design of the study, the collection, analysis and interpretation of data or in writing of the article. Publication costs are funded by Warsaw University of Technology.

Conflict of Interest: none declared.

Data availability

The datasets supporting the results of this article are available upon request in Google Cloud Storage bucket: <gs://biodatageeks/sequila/data/> (for downloading using Google Cloud Storage compatible tool like gsutil) or <https://www.googleapis.com/storage/v1/b/biodatageeks/o?prefix=sequila/data/>. Project source code is publicly available (Apache License) at the GitHub platform at <https://github.com/biodatageeks/sequila>. Cloud deployments documentation and recipes are publicly available at the GitHub platform at <https://github.com/biodatageeks/sequila-cloud-recipes>. Detailed documentation is available on the project site at <https://biodatageeks.github.io/sequila/>.

References

- Ahmad, T. et al. (2021) VC@scale: scalable and high-performance variant calling on cluster environments. *GigaScience*, 10.
- Armbrust, M. et al. (2015) Spark SQL. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, pp. 1383–1394.
- Boettiger, C. (2015) An introduction to Docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49, 71–79.
- Bonfield, J.K. et al. (2019) Crumble: reference free lossy compression of sequence quality values. *Bioinformatics*, 35, 337–339.
- Capuccini, M. et al. (2020) MaRe: processing big data with application containers on apache spark. *GigaScience*, 9.
- Castro, D. et al. (2019) Apache spark usage and deployment models for scientific computing. *EPJ Web Conf.*, 214, 07020.
- Danecek, P. et al. (2021) Twelve years of SAMtools and BCFtools. *GigaScience*, 10.
- Guerriero, M. et al. (2019) Adoption, support, and challenges of infrastructure-as-code: insights from industry. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA. IEEE, pp. 580–589.
- Guo, R. et al. (2018) Bioinformatics applications on apache spark. *GigaScience*, 7, 1–10.
- Heller, J. (2019) Modify data with advanced DML. In: *Pro Oracle SQL Development*. Apress, Berkeley, CA, pp. 191–218.
- Ivanov, T. and Pergolesi, M. (2020) The impact of columnar file formats on SQL-on-hadoop engine performance: a study on ORC and parquet. *Concurr. Comput. Pract. Exper.*, 32.
- Koboldt, D.C. et al. (2012) VarScan 2: somatic mutation and copy number alteration discovery in cancer by exome sequencing. *Genome Res.*, 22, 568–576.
- Koppad, S. et al. (2021) Cloud computing enabled big multi-omics data analytics. *Bioinform. Biol. Insights*, 15, 11779322211035921.
- Krissaane, I. et al. (2020) Scalability and cost-effectiveness analysis of whole genome-wide association studies on Google Cloud platform and Amazon Web Services. *J. Am. Med. Inform. Assoc.*, 27, 1425–1430.
- Lawlor, B. and Sleator, R.D. (2020) The democratization of bioinformatics: a software engineering perspective. *GigaScience*, 9, 1–3.
- Li, H. (2011) A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data. *Bioinformatics*, 27, 2987–2993.
- Li, H. et al.; 1000 Genome Project Data Processing Subgroup. (2009) The sequence alignment/map format and SAMtools. *Bioinformatics*, 25, 2078–2079.
- Liu, Y. et al. (2021) Psi-Caller: a lightweight short read-based variant caller with high speed and accuracy. *Front. Cell Dev. Biol.*, 9, 731424.
- Luo, R. et al. (2020) Exploring the limit of using a deep neural network on pileup data for germline variant calling. *Nat. Mach. Intell.*, 2, 220–227.
- Massie, M. et al. (2013). Adam: genomics formats and processing patterns for cloud scale computing. *Technical Report*, EECS Department, University of California, Berkeley.
- McKenna, A. et al. (2010) The genome analysis toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Res.*, 20, 1297–1303.
- Modi, R. (2021) Deep-dive into terraform. In: *Deep-Dive Terraform on Azure*. Apress, Berkeley, CA, pp. 77–113.
- Niemenmaa, M. et al. (2012) Hadoop-BAM: directly manipulating next generation sequencing data in the cloud. *Bioinformatics*, 28, 876–877.
- Nisbet, A. et al. (2019). Profiling and tracing support for Java applications. In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ACM, New York, NY, USA, pp. 119–126.
- Pedersen, B.S. and Quinlan, A.R. (2018) Mosdepth: quick coverage calculation for genomes and exomes. *Bioinformatics*, 34, 867–868.
- Romanel, A. et al. (2015) ASEQ: fast allele-specific studies from next-generation sequencing data. *BMC Med. Genomics*, 8, 9.
- Sater, V. et al. (2020) UMI-gen: A UMI-based read simulator for variant calling evaluation in paired-end sequencing NGS libraries. *Comput. Struct. Biotechnol. J.*, 18, 2270–2280.
- Sethi, R. et al. (2019) Presto: SQL on everything. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, Macao, China. IEEE, pp. 1802–1813.
- Shah, J. and Dubaria, D. (2019) Building modern clouds: using Docker, Kubernetes & Google Cloud Platform. In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, Las Vegas, NV, USA. IEEE, pp. 0184–0189.
- Shen, Y. et al. (2021) Using vectorized execution to improve SQL query performance on spark. In: *50th International Conference on Parallel Processing*. ACM, New York, NY, USA, pp. 1–11.

- Sipek, M. *et al.* (2020) Enhancing performance of cloud-based software applications with GraalVM and Quarkus. In: *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, Opatija, Croatia. IEEE, pp. 1746–1751.
- Smith, J. *et al.* (2021) Scalable analysis of multi-modal biomedical data. *GigaScience*, **10**.
- Sun, X. *et al.*; CAAPA consortium. (2018) Optimized distributed systems achieve significant performance improvement on sorted merging of massive VCF files. *GigaScience*, **7**.
- Tarasov, A. *et al.* (2015) Sambamba: fast processing of NGS alignment formats. *Bioinformatics*, **31**, 2032–2034.
- Vaillancourt, P. *et al.* (2020). Reproducible and portable workflows for scientific computing and HPC in the cloud. In: *Practice and Experience in Advanced Research Computing*. ACM, New York, NY, USA, pp. 311–320.
- Valentini, S. *et al.* (2019) PaCBAM: fast and scalable processing of whole exome and targeted sequencing data. *BMC Genomics*, **20**, 1–5.
- Wiewiórka, M. *et al.* (2018) SeQuiLa: an elastic, fast and scalable SQL-oriented solution for processing and querying genomic intervals. *Bioinformatics*, **35**, 2156–2158.
- Wiewiórka, M. *et al.* (2019) SeQuiLa-cov: a fast and scalable library for depth of coverage calculations. *GigaScience*, **8**.
- Wiewiórka, M. S. *et al.* (2017) Benchmarking distributed data warehouse solutions for storing genomic variant information. *Database*, **2017**.
- Wilks, C. *et al.* (2021) Megadepth: efficient coverage quantification for BigWigs and BAMs. *Bioinformatics*, **37**, 3014–3016.
- Yuan, D. Y. and Wildish, T. (2020). Bioinformatics application with Kubeflow for Batch Processing in Clouds. In: *Bioinformatics Application with Kubeflow for Batch Processing in Clouds*. Springer, Cham, pp. 355–367.
- Zaharia, M. *et al.* (2010) Spark: cluster computing with working sets. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, Boston, MA, USA, Vol. **10**, p10.