

Enhancing Scalability of Distributed SNPs Calling Pipelines Using Cluster-Driven Partitioning Strategy

Lorenzo Di Rocco^{1,*}, Umberto Ferraro Petrillo¹ and Giorgio Grani¹

¹Dipartimento di Scienze Statistiche, Università di Roma “La Sapienza”, P.le Aldo Moro 5, I-00185 Rome, Italy

Abstract

The genetic composition of individuals within the same species may differ due to mutations in their genomes. Variants calling procedures are pivotal to detect these polymorphisms. Typically, variants calling algorithms use the *De Bruijn* graph data structure for storing the k-nucleotide long strings sequenced from the genomes of multiple individuals. Subsequently, mutations in an individual can be identified by searching for divergent paths (known as *bubbles*) in the corresponding *De Bruijn* graph.

The analysis of a very large *De Bruijn* graph can be very computationally intensive. In a recent proposal, a novel pipeline was proposed to build and analyze very large *De Bruijn* graphs using distributed computing. This novel approach has shown competitive results in the specific case of calling *SNPs*. However, experimental results have shown that scalability is limited due to the overhead of traversing paths scattered across different computers.

In this work, we address one of the main problems that prevent distributed *De Bruijn* graphs from performing well in practice. Namely, we analyze how a bad distributed partitioning of a *De Bruijn* graph may hinder the efficiency of the *SNPs* detection process. Then, we introduce a distributed partitioning strategy based on a hierarchical clustering method able to overcome this problem. We also provide an experimental analysis showing how this approach allows for a relevant performance boost of the considered pipeline and ensures for better scalability.

Keywords

Graph Partitioning, Distributed Computing, Computational Genomics

1. Introduction

Advances in New Generation Sequencing (NGS) technologies have led to an explosion in the amount of genomics data at affordable costs. Because of this massive production of biological datasets, pivotal tasks in Bioinformatics have become computationally expensive.

One of the most relevant tasks is *variants calling*. We use this term to denote a procedure aimed at detecting variants between the genomes of two or more different individuals. *Reference-based* algorithms are commonly used to detect the presence of mutations with respect to the average genome of the corresponding species. However, high-quality reference genomes are not available for all species and for all applications. For these reasons, it is useful to consider


ITADATA2023: The 2nd Italian Conference on Big Data and Data Science, September 11–13, 2023, Naples, Italy


*Corresponding author.

✉ lorenzo.dirocco@uniroma1.it (L. Di Rocco); umberto.ferraro@uniroma1.it (U. Ferraro Petrillo);

g.grani@uniroma1.it (G. Grani)

ORCID 0000-0001-8744-7048 (L. Di Rocco); 0000-0002-4308-5126 (U. Ferraro Petrillo); 0000-0001-6049-0062 (G. Grani)

 © 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

reference-free algorithms. These allow comparison of individuals by evaluating reads sequenced from the corresponding genomes, without relying on an average species sequence.

The state-of-the-art reference-free methods are usually based on the exploration of a graph-based data structure, called *De Bruijn* (DB) graph, that is built from an input collection of reads. The presence of a polymorphism is then detected by finding some specific types of divergent paths in the corresponding DB graph. However, the computational and memory requirements of these algorithms can be prohibitive, making them unsuitable for processing large DB on a single workstation. Some alternative methods that have emerged recently therefore use distributed computing to develop reference-free pipelines for variants calling, able to take advantage of the large availability of memory and computational resources of a computer cluster.

Despite the theoretical advantages, analyzing a DB graph on a distributed architecture poses significant challenges. For example, the data traffic required to explore a region of the graph where each vertex of a path is on a different node of the underlying distributed system can create excessive overhead that negatively affects algorithm performance and scalability.

In this work, we focus on the specific problem of reducing the communication overhead to pay for exploring distributed DB graphs by considering a recent distributed pipeline presented in [1], that aims at calling a specific type of polymorphisms, namely the Single Nucleotide Polymorphisms (*SNPs*). The solution we present is based on a cluster-driven partitioning strategy used in the initial creation of a DB graph. We expect that this will significantly reduce the amount of data exchanged between nodes during the invocation of *SNPs*, which should have a positive impact on the efficiency and scalability of the pipeline.

We also present the results of an experimental analysis performed on a real dataset to compare our improved partitioning scheme with the default partitioning strategy used by [1].

2. Background

2.1. De Bruijn graphs

The DNA structure consists of two antiparallel strands comprising nucleotide bases, commonly represented by the alphabet $\Sigma = \{A, C, G, T\}$. The strands run in opposite directions, and the bases in the corresponding positions are paired according to specific rules (A-T and C-G). Consequently, a genomic sequence is usually encoded as a single string of characters drawn from the alphabet Σ , as the reverse strand is easily obtainable. However, modern sequencing technologies do not return the entire genome in one continuous sequence. Instead, they generate huge amounts of nucleotide fragments, called *reads*. These reads are randomly sequenced from both strands, and the coverage level determines the number of reads associated with a particular genomic region.

Consequently, reconstructing a genome becomes a complex task due to the chaotic reads samples that need to be assembled. The assembly phase is extremely expensive from the computational viewpoint since it needs to face the time complexity of computing the matching score for each pair of reads.

Assembly techniques based on k-mers and the DB graph have revolutionized genomic analysis. The k-mer-based approach involves splitting the genomic reads into smaller overlapping

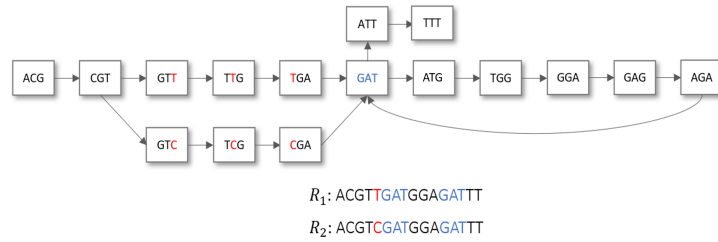


Figure 1: The DB graph obtained from two sequenced reads (R_1 and R_2), when $k = 3$. The set of vertices represent all the distinct 3-mers extracted from R_1 and R_2 , while the edges connect the overlapping 3-mers. The *SNP* (marked in red) between R_1 and R_2 generates two parallel paths connecting CGT with GAT. The substring GAT occurs twice, determining a loop in the graph and two branching paths.

subsequences of length k (called k -mers) and building the corresponding DB graph. The vertices of a DB graph represent the distinct k -mers extracted from the reads sample, while the edges connect the k -mers that overlap for $k - 1$ characters. The DB graph provides a compact and efficient representation of the genome, and traversing its path of overlapping sequences enhances an efficient reconstruction of the original genomic sequence.

DB graphs play also crucial role in variants calling. Variants calling refers to the process of identifying genetic variations between different individuals, such as single nucleotide polymorphisms (SNPs) and insertions/deletions (indels), in a given genomic dataset. The structure of the DB graph enables efficient and accurate detection of variants. Indeed, when dealing with sequencing data referring to two or more individuals, the presence of polymorphisms generate divergent paths, called *bubbles*. There exist different type of bubbles according to corresponding polymorphisms. In the case of isolated SNPs, we have simple bubble. Figure 1 shows an example. In real world scenario the topology of graph is extremely complex and the detection of the divergent is not straightforward.

2.2. Large-scale variants calling

DB graphs tend to become very large and, consequently, difficult to store, when based on genomic datasets returned by modern sequencing technologies. A possible solution to this problem consists of introducing compressed data structures for storing a DB graph. This problem is addressed in [2, 3], where a concise representation of a colored DB graph is obtained by using data structures for string compression and indexing, as the positional Burrows-Wheeler transform [4].

Further solutions take into account a probabilistic representation of the DB graph. Among the proposed techniques, *DiscoSnp* [5] is one that outperforms many others in terms of space and time efficiency, when used to retrieve isolated SNPs on genomes of complex organisms. It features an efficient storage layout where only the nodes of an input DB graph are stored using a cascade of Bloom Filters [6]. The edges are found by properly querying these filters to obtain

paths corresponding to isolated *SNPs*, even though the Bloom Filters may return non existing paths due to false positive matches.

Despite the potential advantages, the usage of distributed computing for large-scale variants calling analysis still remains a relatively unexplored field. To the best of our knowledge, [1] is the only pipeline that leverages a distributed representation of a DB graph to perform *SNPs* calling.

2.3. Distributed Computing

Computations on distributed systems enable the processing of huge collections of data. In distributed computing architectures, a number of computers (called nodes) work together to solve a single problem. The nodes coordinate their hardware resources using messages exchanged over a network.

2.3.1. MapReduce

The *MapReduce* (MR) paradigm [7] is a programming model designed for implementing distributed algorithms. It considers an input dataset of key-value pairs distributed across a computer cluster and provides a flow that combines two types of functions:

- *map functions*. They process each key-value pair to obtain a new (possibly empty) set of key-value pairs.
- *reduce functions*. They first collect on a same node all pairs sharing the same key and, then, they aggregate all values in this collection, returning a (possibly empty) set of key-value pairs

Apache Spark [8] is the standard framework for developing *MapReduce* algorithms to be executed on a distributed computing architecture. *Resilient Distributed Datasets* (RDDs) are the core data structures of Spark. RDDs are an abstract representation of the key-value pairs distributed across the nodes of the distributed system.

2.3.2. Apache Spark for distributed graph processing

Apache Spark also provides libraries (built on top of RDDs) dedicated to specific tasks. The *GraphX* API [9] is designed for large-scale graph processing on a Spark cluster. This API introduces a distributed representation of a graph built on top of two RDDs, storing respectively the vertices and the edges of a graph, with their associated properties. In details, each vertex is associated with an automatically-generated unique id (*vid*) and (optionally) a set user-defined attributes. The edges are modeled as triplets containing the source vertex id, the destination vertex id, a virtual partition identifier (*pid*) and (optionally) a set user-defined attributes. Vertices and edges are scattered across the nodes of a computer cluster, according to their *vid/pid*. The *GraphX* API employs a default partition strategy and generates the so-called *VertexMap* RDD that maps each vertex to the list of partitions containing its incident edges. Leveraging on the vertices and the edges identifiers, *GraphX* allows also to consider user-defined graph partition strategies that may minimize the traffic data and balance the workload of the computing nodes.

GraphX allows to traverse and to analyze a graph, by means of algorithms written using a special-purpose message-passing paradigm.

3. A MapReduce Pipeline for Isolated SNPs Calling

The distributed pipeline we consider in this work has been recently proposed in [1]. It has been designed according to the *MapReduce* paradigm (see Section 2.3.1) and it is a distributed reformulation of the *DiscoSnp* algorithm [5], with some relevant improvements. We recall that *DiscoSnp* uses a cascade of Bloom filters to store the DB graph. This allows to maintain very large graphs in a small amount of memory, but it has the disadvantage of leading to the generation of paths that were not present in the original graph (i.e., *chimeric sequences*) and, therefore, have to be eliminated in a subsequent step.

In a distributed system, the amount of available memory is usually much larger, since we can leverage on the computational resources of all nodes of the system. Therefore, in the approach presented in [1], the input DB graph is explicitly represented, with no compression at all, thanks to a distributed representation, thus avoiding the creation of useless *chimeric sequences*.

Specifically, the proposed algorithm is formulated in three steps, each made of a sequence of distributed transformations. In the first step, a DB graph \mathcal{G} is created from an initial distributed collection \mathcal{C} of reads. In the second step, \mathcal{G} is processed by a distributed algorithm that searches for all simple bubbles. In the third step, all isolated bubbles, i.e., the *SNPs*, are returned in the output. More details on the algorithm follow.

3.1. Step 1: Graph Creation

Given two FASTA/FASTQ files, each containing a set of reads extracted from two different individuals, their contents are loaded in memory using the FASTdoop library [10]. Then, $(k + 1)$ -mers are extracted from each read and stored in a RDD distributed data structure. Finally, a reduce transformation is executed to count the frequency of each distinct $(k + 1)$ -mer. The results are stored in a new RDD, containing all the $(k + 1)$ -mers that have been found in the input reads collection, with their associated overall frequencies.

A map transformation is used to filter out $(k + 1)$ -mers with low coverage. Then, after merging all samples, a DB distributed graph is created using the GraphX library (see Section 2.3.2). Next, we extract consecutive k -mers from the $(k + 1)$ -mers to feed the distributed collection of vertices of the DB graph. Then, $(k + 1)$ -mers are added to the distributed collection storing the edges of the DB graph. The edges connect the vertices corresponding to the two consecutive k -mers. This process is also performed for the backward strand of each $(k + 1)$ -mer.

3.2. Step 2: Centrality Indices Evaluation

In this step, two centrality indices telling the number of incoming and outgoing edges for each vertex v , respectively denoted *inDegree* and *outDegree*, are evaluated.

To do this, for each vertex v , the number of triples containing v in the distributed representation of \mathcal{G} is counted, distinguishing the cases where v is the source vertex from those where v is the

destination vertex. Then, for each vertex, these counts are computed by a distributed reduce transformation to obtain the corresponding centrality indices.

3.3. Step 3: Simple Bubbles Detection

In this step, a distributed algorithm is executed over \mathcal{G} to find all paths that represent simple bubbles.

Initially, vertices with an outDegree greater than 1 are considered as *starting* vertices and are marked as *enabled*. In the first iteration, each enabled triplet sends a *path-building* message to its adjacent vertices, including information about the current path (which initially contains only the starting point) and the iteration number.

In subsequent iterations, the triplets containing vertices that received messages in the previous iteration are enabled. When a vertex receives a path-building message from a neighbor, it updates the received message by adding its own identity and increasing the iteration number. The vertex stores a copy of the iteration number in its state and broadcasts the updated message to its adjacent vertices.

To handle branching bubbles, the algorithm modifies the transmission of path-building messages. In the first $k-1$ iterations, the message is only transmitted to destination vertices with one incoming edge and one outgoing edge. Once the number of iterations reaches k , the messages are forwarded without checking the degrees of the destination vertex. This change is because at iteration k , the destination vertex is expected to be the end vertex of the bubble, allowing multiple incoming and outgoing edges.

At the end of the execution, after running k iterations, the algorithm identifies vertices with iteration number $k + 1$ that have received two messages. These messages contain paths starting from the same vertex and with the same length. These vertices represent terminal nodes of sequence pairs containing simple bubbles and are considered as isolated *SNPs*. They are encoded as a distributed collection of string pairs.

4. Our Proposal

4.1. Motivation

When using a distributed approach, one expects that the solution time of a given problem could be reduced by just employing more computational resources. However, the ability to efficiently exploit the computational capability of a distributed system is not always a given. This is either due to the inherent non-parallelism of the problem being solved, or to issues involving the algorithm itself, its implementation and the way it interacts with the underlying distributed system. This is the case of the pipeline presented in [1]. As shown in Figure 2, the experiments therein presented have shown a promising level of scalability when exploiting an increasing number of computing units. However, the reduction in execution times is gradually less and less pronounced.

A more detailed investigation revealed that this lack of scalability is mainly due to the suboptimal strategy GraphX uses for partitioning the vertices of the DB graph across different

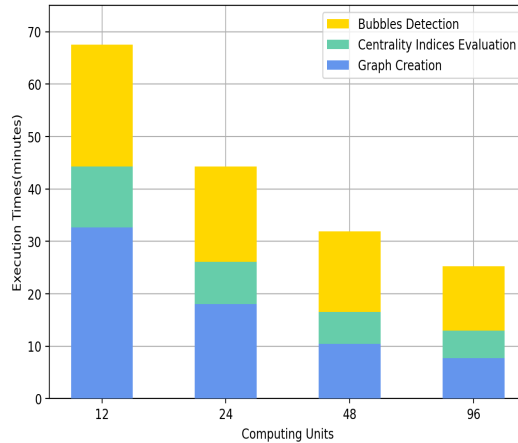


Figure 2: Execution times required by [1] for isolated *SNPs* calling on a dataset including 32 GB of reads sequenced from the chromosome 7 of two human individuals, using an increasing number of computing units.

nodes of a distributed system, resulting in a very high number of *x-cross* paths, i.e. paths of a distributed graph where vertices are placed over two or more nodes of the distributed system.

In this scenario, the advantages of adding further computing nodes may be burdened by the communication overhead introduced by the message-passing mechanism required to explore the distributed graph. This overhead includes both the exchange of data between computing units within a node and across different nodes. The latter case is significantly more expensive in terms of network communication times and scalability level.

Our work focuses on assessing the positive impact on the performance of the considered algorithm coming from the adoption of a partitioning strategy different from the standard one available with GraphX. Our expectation is that the usage of an improved scheduling strategy would significantly reduce, when not eliminating at all, the number of *x-cross* paths. In turn, this would lead to a significant reduction in the communication overhead between different computing nodes and/or computing units, thus allowing for shorter overall execution times. To achieve this, our proposal leverages a hierarchical clustering approach to group reads referring to the same genomic region, enabling an intra-cluster solution for bubbles detection.

4.2. Cluster-driven Partitioning Strategy

The strategy we propose to improve the partitioning of a DB graph over a distributed system is based on the assumption that clustering the reads according to an appropriate similarity metric isolates specific genomic regions, allowing to perform the bubble detection procedure mostly within them. Consequently, assigning these clusters to different independent computing units for parallel processing is expected to require less communication overhead than using the standard partitioning strategy, thus enhancing the scalability of the DB graph exploration in a distributed environment.

It requires two steps. In the *Hierarchical Clustering* step, the collection of input reads about

two distinct individuals are clustered using a bottom-up approach and a greedy stop criterion. In the *Clusters Binning* step, the clusters returned in the previous step are grouped into B distinct bins scattered across the computational units of the distributed architecture. The generation of the bins is driven by the application of the longest processing time first rule (i.e., *LPT* rule)[11, 12], to avoid overloading certain nodes. In the following, we will explain the proposed partitioning strategy in more detail.

4.2.1. Hierarchical Clustering

Given an input set of reads sequenced from two individuals, we use a hierarchical bottom-up algorithm to group those reads that are likely to cover the same genomic region.

An algorithm of this type implements an iterative procedure that progressively merges similar elements into larger clusters to eventually obtain a *dendrogram*, i.e., a hierarchical tree-like structure representing the clustering relationships. In the initialization step, each read is considered as a cluster with a single element. Then, in the i -th iteration, for each cluster, the read closest to the others in the same cluster is selected as representative. Then, the pairwise similarity matrix between all clusters is calculated and the two closest clusters are merged. This process continues until obtaining a single large cluster that contains all the reads in the input dataset.

Creating the entire dendrogram can be very computationally intensive when working with large datasets. Moreover, solutions with a small number of clusters should be avoided, as our strategy aims to detect clusters consisting of very similar reads that refer to the same genomic region. To achieve this, we incorporated a greedy stopping criterion into the algorithm. Namely, our hierarchical iterative process stops when the similarity between the two clusters to be merged at the i -th iteration is less than a certain threshold c . In this way, the choice of the number of clusters is also automated without the need for a global evaluation of the dendrogram. If the threshold c is sufficiently large, hierarchical clustering yields a large number of clusters containing a relatively small number of reads. However, if c is too large, reads related to the same genomic section may be assigned to different clusters, causing many bubbles to be missed during the detection step.

4.2.2. Choice of a similarity measure

We recall that a genomic read is simply a string of characters from the alphabet $\Sigma = \{A, C, G, T\}$. Our clustering framework, therefore, requires a similarity measure that can capture the underlying similarity patterns between strings and is not extremely sensitive to slight variations in characters. This helps to ensure that reads sequenced from multiple individuals covering the same genomic regions are clustered together, despite the presence of *SNPs* in the same cluster.

In this work, we consider the *Jaccard index*, also known as *Jaccard similarity coefficient*. This measure is widely used in computational genomics for assessing similarity between nucleotide strings and for clustering [13, 14]. The Jaccard index quantifies the similarity between two reads with respect to their shared k -mers. To calculate this coefficient, it is required to extract the corresponding sets of k -mers from a pair of reads and calculate the percentage of common k -mers. Mathematically, the Jaccard index can be expressed as follows:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

where:

- A and B represent the sets of k-mers extracted from the two genomic reads;
- $|A|$ and $|B|$ represent, respectively, the number of distinct k-mers in A and B ;
- $|A \cap B|$ represents the number of shared k-mers between A and B ;
- $|A \cup B|$ represents the total number of unique k-mers in the union between A and B .

The Jaccard index ranges between 0 and 1. The more it is larger, the more the reads are similar. When it is equal to 1, the reads are identical. Since the Jaccard index evaluates the presence or the absence of shared k-mers, it is robust to small variations in the reads being compared.

4.2.3. Binning the clusters

The clusters returned in the previous step need to be distributed among the computational units of the distributed system to perform bubble detection in parallel. In this way, each computing unit processes a bin of clusters according to the available resources. However, the number of reads changes within the cluster, and some clusters may be overloaded due to the presence of highly repetitive genomic regions. For this reason, the binning strategy is controlled by an appropriate scheduling model, to ensure a balanced workload on the computational units.

We consider this problem as a scheduling problem with identical parallel machines and no preemption. The objective is to minimize the maximum completion time (makespan) by assigning each of the n jobs, characterized by their respective processing times $\{p_j, j = 1, \dots, n\}$, to one of the m parallel identical machines. In our context, these jobs correspond to the tasks associated with building and exploring the local DB graph belonging to a cluster of reads. The processing time of each job is estimated based on the number of k-mers extracted from the reads within the same cluster. The machines, on the other hand, represent the available computational units. This problem, referred to as $P||C_{max}$, is known to be NP-hard and can be formulated mathematically as follows:

$$\min C_{max} \tag{1}$$

$$C_{max} \geq \sum_{j=1}^N p_j x_{ij} \quad \forall i \in \{1..m\} \tag{2}$$

$$\sum_{i=1}^m x_{ij} = 1 \quad \forall j \in \{1..N\} \tag{3}$$

$$x_{ij} \in \{0, 1\} \quad \forall i \in \{1..m\} \quad j \in \{1..N\} \tag{4}$$

where C_{max} is the *makespan*, N is the number of jobs, m is the number of machines, p_j is the processing time of job j , x_{ij} is the decision variable. The value is 1 if the job j is assigned to machine i , otherwise it is 0. Given the complexity of this problem, many heuristic solutions have been proposed in literature, as the *LPT* rule. It involves sorting jobs in a non-increasing order based on their processing times and assigning each job iteratively to the machine that currently has the minimum completion time. The completion time of a machine is determined by summing the processing times of all the jobs assigned to that particular machine.

5. Experimental Analysis

We conducted an experimental analysis to assess the quality of the proposed clustering-driven partitioning strategy and evaluate its impact on the scalability.

5.1. Dataset

We downloaded the FASTA file corresponding to chromosome 7 of the average human genome assembly GRCh37.p13 [15]. From this FASTA file, we simulated a sample of reads for our experiments. Additionally, we used SAMtools [16] to extract the sequencing data referring to chromosome 7 of the European individual NA12878 from the 30x downsampled alignment provided by the Genome-in-a-Bottle Consortium [17]. Then, we built a distributed DB graph from the two sets of reads fixing $k = 31$, as described in Section 3.1.

The individual NA12878 is accompanied by a VCF file containing the set of high-confidence variants with respect to the average GRCh37.p13 genome. From this VCF file, we selected 10,000 SNPs for our analysis. Thereafter, we extracted the portions of the DB graph corresponding to these SNPs, to compare the GraphX default partitioning strategy with the one outcoming from our cluster-driven partitioning strategy. From now on, we refer to the filtered DB graph used in our analysis as DB1e4.

5.2. Computing Environment

We conducted our experiments on a high-performance computing system that consists of eight computing nodes. Each node runs on the Linux operating system and is equipped with two AMD Epyc 7452 processors, 64 compute cores, and 256 GB of RAM. For our experiments, we employed the 3.1.3 Apache Spark version and the 2.7 Apache Hadoop version.

5.3. Evaluating default partitioning strategy

First, we evaluated the GraphX default partitioning strategy on DB1e4. Thanks to the information provided by the VCF file, containing mutations about the individual NA12878, with respect to the GRCh37.p13 version of the average human genome, we easily identified the bubbles occurring in the graph. For each bubble, we determined the number of computing units that need to communicate in order to traverse each edge of the bubble. Since we set $k = 31$, each bubble may be split into up to 31 different computing units.

In this case, the histogram shows that there is no bubble that can be traversed by staying on a single computing unit. Instead, it is required to involve at least two computing units while, the most frequent case, requires the involvement of three computing units.

5.4. Scalability

We compared the time-performance of the distributed pipeline described in Section 3, when using the GraphX default partitioning strategy against version based on our cluster-driven partitioning scheme, in terms of scalability.

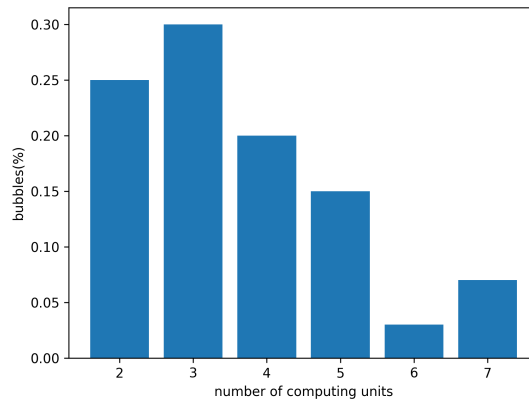


Figure 3: The histogram illustrates the distribution of the bubbles, in terms of the number (in percentage) of computing units to involve for traversing them, according to the default GraphX partitioning strategy.

First, we performed the *SNPs* calling procedure on DB1e4 using the aforementioned pipeline and the default GraphX partitioning strategy, while increasing the number of computing units. Then, we repeated the same experiment, but using our hierarchical clustering framework and binning strategy. In this last case, we considered for each run a number of bins that was 4 times the number of allocated computing units. The resulting bins were then loaded into memory on the distributed computing architecture, to perform the bubble detection procedure.

Figure 4 compares the scalability deriving from the application of the two partitioning approaches, highlighting how our proposed strategy significantly improves the ability to leverage on an increasing number of computing resources.

6. Conclusions

In this work, we focused on a performance bottleneck affecting Spark-based distributed pipelines used for *SNPs* detection and due to the bad performance of the default partitioning strategy employed by the Spark framework when creating distributed graphs. Thus, we have proposed an improved partitioning strategy, based on a hierarchical clustering algorithm. According to our experimental results, the proposed strategy is able to significantly improve to enhance the scalability of the considered distributed computing pipeline for *SNPs* calling. In future directions, it will be pivotal to explore different similarity measures and algorithms for genomic reads clustering, as the Jaccard index and hierarchical clustering algorithms can be computationally expensive when dealing with huge-scale analyses. Furthermore, ongoing research is evaluating strategies for detecting more complex mutations and dealing with genomic regions characterized by a high mutation intensity, whose corresponding reads may not fall within the same cluster.

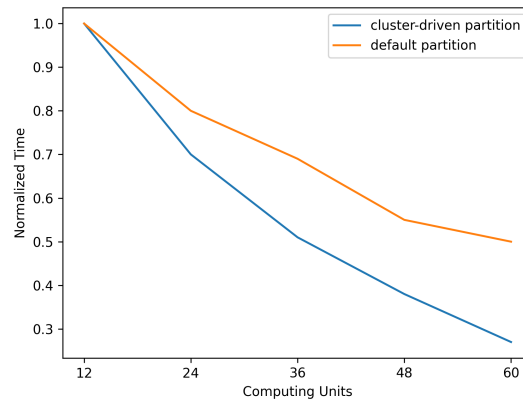


Figure 4: Time performances of the *SNPs* calling procedure executed on the DB1e4 graph, when considering the GraphX default partitioning and the cluster-driven partitioning strategies. The execution times are normalized and reported as a function of the employed computing units.

Acknowledgments

The authors would like to thank the Department of Statistical Sciences of University of Rome - La Sapienza for computing time on the TeraStat cluster. This work was partially supported by Università di Roma - La Sapienza Research Project 2021 “Caratterizzazione, sviluppo e sperimentazione di algoritmi efficienti”. It was also supported in part by INdAM – GNCS Project 2023 “Approcci computazionali per il supporto alle decisioni nella Medicina di Precisione”.

References

- [1] D. Geiger, C. Meek, Structured variational inference procedures and their realizations (as incol), in: Proceedings of Tenth International Workshop on Artificial Intelligence and Statistics, The Barbados, The Society for Artificial Intelligence and Statistics, 2005.
- [2] M. D. Muggli, A. Bowe, N. R. Noyes, P. S. Morley, K. E. Belk, R. Raymond, T. Gagie, S. J. Puglisi, C. Boucher, Succinct colored de bruijn graphs, *Bioinformatics* 33 (2017) 3181–3187.
- [3] G. Holley, P. Melsted, Bifrost: highly parallel construction and indexing of colored and compacted de bruijn graphs, *Genome biology* 21 (2020) 1–20.
- [4] R. Durbin, Efficient haplotype matching and storage using the positional burrows–wheeler transform (pbwt), *Bioinformatics* 30 (2014) 1266–1272.
- [5] R. Uricaru, G. Rizk, V. Lacroix, E. Quillery, O. Plantard, R. Chikhi, C. Lemaitre, P. Peterlongo, Reference-free detection of isolated snps, *Nucleic acids research* 43 (2015) e11–e11.
- [6] K. Salikhov, G. Sacomoto, G. Kucherov, Using cascading bloom filters to improve the memory usage for de bruijn graphs, in: International Workshop on Algorithms in Bioinformatics, Springer, 2013, pp. 364–376.

- [7] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, *Communications of the ACM* 51 (2008) 107–113.
- [8] Apache Software Foundation, Apache Spark, (Available from: <http://spark.apache.org>), 2016.
- [9] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica, Graphx: A resilient distributed graph system on spark, in: *First international workshop on graph data management experiences and systems*, 2013, pp. 1–6.
- [10] U. Ferraro Petrillo, G. Roscigno, G. Cattaneo, R. Giancarlo, Fastdoop: a versatile and efficient library for the input of fasta and fastq files for mapreduce hadoop bioinformatics applications, *Bioinformatics* 33 (2017) 1575–1577.
- [11] R. L. Graham, Bounds on Multiprocessing Timing Anomalies., *SIAM Journal on Applied Mathematics* 17 (1969) 416–429.
- [12] L. Amorosi, L. D. Rocco, U. F. Petrillo, Scheduling k-mers counting in a distributed environment, in: *Optimization in Artificial Intelligence and Data Sciences: ODS, First Hybrid Conference, Rome, Italy, September 14-17, 2021*, Springer, 2022, pp. 73–83.
- [13] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, E. Solomonik, Communication-efficient jaccard similarity for high-performance distributed genome comparisons, in: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2020, pp. 1122–1132.
- [14] Z. Rasheed, H. Rangwala, A map-reduce framework for clustering metagenomes, in: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, IEEE, 2013, pp. 549–558.
- [15] D. M. Church, V. A. Schneider, T. Graves, K. Auger, F. Cunningham, N. Bouk, H.-C. Chen, R. Agarwala, W. M. McLaren, G. R. Ritchie, et al., Modernizing reference genome assemblies, *PLoS biology* 9 (2011) e1001091.
- [16] P. Danecek, J. K. Bonfield, J. Liddle, J. Marshall, V. Ohan, M. O. Pollard, A. Whitwham, T. Keane, S. A. McCarthy, R. M. Davies, et al., Twelve years of samtools and bcftools, *Gigascience* 10 (2021) giab008.
- [17] J. M. Zook, B. Chapman, J. Wang, D. Mittelman, O. Hofmann, W. Hide, M. Salit, Integrating human sequence data sets provides a resource of benchmark snp and indel genotype calls, *Nature biotechnology* 32 (2014) 246–251.