2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

ArrOW: Experiencing a Parallel Cloud-based *De Novo* Assembler Workflow

Kary Ocaña[1], Thaylon Guedes[2], Daniel de Oliveira[2]
[1]National Laboratory of Scientific Computing - LNCC, Petrópolis Rio de Janeiro - Brazil
[2]Institute of Computing, Fluminense Federal University - UFF, Niterói - Rio de Janeiro - Brazil
karyann@lncc.br; thaylongs@id.uff.br; danielcmo@ic.uff.br

***Abstract*—Advances in next generation sequencing technologies has resulted in the generation of unprecedented volume of sequence data. DNA segments are combined into a reconstruction of the original genome using computer software called genome assemblers. Therefore, assembly now presents new challenges in terms of data management, query, and analysis due the huge number of read sequences and computing intensive CPU-memory algorithms. This restriction reduces the chances to uniformly cover space for exploring statistics, *k-mer*, software or eukaryotic genomes assembly. To address these issues, we present ArrOW, a cloud-based *de novo Assembly clOud Workflow* that explores the potential of provenance analytics and parallel computation provided by scientific workflow management systems as SciCumulus. We evaluate the overall performance of ArrOW using up to 256 cores in the Amazon AWS cloud. ArrOW reaches improvements up to 88.3% executing 1,000 reads of genomics datasets. We also highlight how data provenance analytics improved the efficiency for recovering assembling features of genomes.**

Keywords-component; workflow; cloud; assembly

I. Introduction

The current genomic revolution was possible due to joint advances in Next Generation Sequencing (NGS) and genomics computational approaches. The close interaction of biology, chemistry, engineering, and computer science is perhaps most apparent in the development of sophisticated genomics computational software called assemblers. Assemblers aim to reconstruct genome sequences from the many small segments of DNA that can be read by modern DNA sequencing machines. Genome size is a factor associated with the complexity in assembly. Bacteriophages and viruses range from a few thousand base pairs (bp) to several hundred Kb, bacterial genomes range from 0.5 to 10 Megabase pairs (Mb), eukaryotic genomes are diverse, from 10 Mb in fungi to more than 100,000 Mb in plants. The human genome comprises more than 4 Gigabase pairs (Gb). However, even using complex mathematical optimizations [1], genome assembly is still computationally intractable.

The strategy adopted by genome assemblers can be widely divided into comparative assembly and *de novo* assembly. Actually, *de novo* assemblers are most commonly used to assembling. Two common types of *de novo* assemblers are greedy algorithms [2] and based on Bruijn graph [1]. In terms of complexity and time requirements, *de novo* is order of magnitude slower and more memory intensive than mapping assemblies. This is mostly due to the fact that the assembly algorithm needs to compare every read with every other read (an operation that has a naive time complexity of $O(n^2)$.

Although some scientists already develop automated experiments for assembling data analysis, they are not mature yet [3]–[5]. Assembly experiments are based on complex computer simulations that consume and produce large datasets and allocate huge amounts of computational resources. To help scientists in managing resources involved in large-scale simulations, scientific workflows are gaining much interest. A scientific workflow is an abstraction that structures steps of a scientific experiment as a graph of activities, in which nodes correspond to data processing activities and edges the dataflow between them [6]. Scientific Workflow Management Systems (SWfMS) allow for defining, executing, and monitoring workflow execution. These engines distribute several concurrent activity executions in a High Performance Computing (HPC) environment while capturing provenance data [7] *i.e.,* historical information of the workflow execution.

The complexity of designing an assembly workflow is also related as the task of choosing the set of programs and parameters, the most adequate to assembly process. The complexity of its implementation is related to the management of thousands of combinations of DNA fragments, parameters variability of *k-mers*, and results. SWfMS can be used to control DNA fragments and parameters combination automatically as a parameter sweep problem. Since assembly workflows may execute for weeks they require HPC resources and technologies. Clouds are becoming a viable alternative to traditional clusters and grids as they have demonstrated applicability to a wide-range of problems in several scientific domains [8]. Cloud environments like Amazon AWS [9] ease the deployment of experiments and data as services.

As aforementioned, processing data in parallel is an open issue and only part of the solution at exploring large assembly datasets. There are assembly workflows that explore parallel processing with different approaches, but all of them limit the number of data or parameters in each execution [3], [10] and none of them vary the parameter and program for each read sequences input. Tracking these variations along results is very complex but needed in assembly analyses. For instance, for each assembler may be needed to perform several workflow executions using different *k-mers*.

This paper addresses several problems of designing a HPC assembly workflow with provenance data analytics at large-scale. We present the Assembly clOud Workflow (ArrOW) using SciCumulus [11] Cloud SWfMS. For genomics data, ArrOW invokes Ray, Velvet, and MetaVelvet. Otherwise, for metagenomics, it invokes only MetaVelvet. Finally, results of ArrOW are analyzed with the help of provenance database by submitting high level database analytical queries. We evaluate ArrOW with Illumina metagenomics data (human nares) and

978-1-5386-5555-9/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPSW.2019.00039

IEEE computer society

present a performance analysis of ArrOW parallel executions using up to 256 cores in Amazon AWS. The results show that ArrOW is capable of processing up to 1,000 read sequences with significant performance improvements. The highest performance gain in ArrOW reduces execution time from 1.5 days (8 cores) to 3.4 hours (256 cores).

This paper is organized as follows. Section II discusses related work. Section III presents background on assembly. Section IV describes the specification of the ArrOW workflow and presents its implementation using SciCumulus SWfMS. Section V shows the experimental results and Section VI concludes the paper and points out future work.

II. RELATED WORK

Relevant works of computational challenges for whole-genome sequencing, assembly, and annotation are surveyed by El-Metwally *et al.* [12] and Ekblom and Wolf [13]. The viral genome assembly pipeline VirAmp [14] connects several existing programs via the web-based platform Galaxy and is available in Amazon Elastic Cloud disk image. However, no computational performance analyses were performed. Several workflows, such as iMetAMOS [10] and A5 [3] were proposed to identify and select the best assembler; however, they focus on prokaryote genome assemblies, which demands manageable and easier computational resources. RAMPART [4] uses a configurable SWfMS for *de novo* genome assembly that executes combinations of third-party tools and settings with good results for eukaryotic genomes. RAMPART supports HPC technologies, shared memory systems, and Platform Load Sharing Facility and Portable Batch System schedulers with plans to support Sun Grid Engine in future. However, it does not provide workflow provenance-based analysis for fault tolerance or re-execution of failed activities.

META-pipe [5] pipeline provides integration with identity provider services, distributed storage, Galaxy workflows, Stallo job scheduler, and interactive data visualizations. The scalability and performance have evaluated on distributed compute and storage resources. META-pipe 1.0 has several limitations as a scalable bioinformatics service. Their custom pipeline framework has no support for provenance data nor failure handling. The re-implemented META-pipe 2.0 uses Apache Spark and can take advantage of novel HPC features.

III. BACKGROUND OF SEQUENCE ASSEMBLY

Sequence assembly refers to aligning and merging many fragments of a much longer target DNA sequence in order to reconstruct the target sequence. Due to the randomness of the fragmentation process of the DNA, the individual fragments could be expected to overlap one another, and these overlaps could be recognized by comparing the DNA sequences of the corresponding fragments. This process is time consuming and labor intensive. However, that even with mathematical formulations [1], genome assembly can be shown to be computationally intractable due to the issue of repeats and the complexity they introduce in the assembly process.

The simplicity of the shotgun sequencing process captured the interest of mathematicians and computer scientists and led to the rapid development of the theoretical foundations for genomics sequence assembly. Lander and Waterman [15] estimated the size and number of contiguous genomic segments that can be reconstructed from a set of sequence reads as a function of just three parameters: length of reads, coverage of genome, and minimum overlap between two reads that can be effectively detected by an assembler.

The computational complexity of the assembly problem is formalized as an instance of the shortest common superstring problem for finding the shortest string that encompasses all reads as substrings. Since finding the correct solution may require to explore an exponential number of possible solutions, the genome sequence assembly can become computationally intractable. The complexity of sequence assembly depends on the ratio between the size of sequence reads and repeats; when reads are longer than repeats, the assembly problem can be solved, whereas when reads are shorter than repeats. Efforts to formalize and understand genome sequence assembly have led to increasingly complex explanations of the problem, which fall into several broad paradigms, outlined in more detail following.

A. Greedy-Based Approaches

Greedy algorithm is one of the simplest strategies for assembling genomes [2]. The process starts by joining the two reads that overlap the best; then repeats this process until a predefined minimum quality threshold is reached. As a result, nascent assembled sequences (contigs) grow through either the addition of new reads or a joining with previously constructed contigs. Many of the early genome sequence assemblers relied on such a greedy strategy: Phrap used by the Human Genome Project, TIGR Assembler, and CAP series of tools to reconstruct transcriptomes. Despite its tremendous early successes, the greedy strategy has a severe limitation since it cannot effectively handle repeated genomic regions.

B. Graph-Based Approaches

Graph-based assembly models represent sequence reads and their inferred relationships to one another as vertices and edges in the graph. Walks through the graph describe an ordering of reads that can be assembled together. The assembler tries to find a walk that best reconstructs the underlying genome while avoiding generating misassemblies by taking erroneous paths caused by repeats.

a. OLCgraphs

OLC genome assemblers follow three main stages, overlap, layout, and consensus. First, overlapping pairs of reads are detected. Second, the graph is constructed, and an appropriate ordering and orientation (layout) of the reads are found. Finally, a consensus sequence (contig) is computed from the ordered and oriented reads. This approach has been used successfully by Celera to assembly the human genome.

Overlap requires significant compute time. Naively, one could simply compare all pairs of reads using dynamic programming to check whether each pair has a significant overlap. Such an algorithm requires $O(n^2)$ time, where n is the total number of sequenced bases. To accelerate overlap detection, an index is constructed to maps *k-mers* to the list of reads containing the *k-mer* (k in range of 16–24 bases). Layout tries to generate unitigs (collections of reads assembled) [16]. The assembler removes low-quality sequence reads and

overlaps that are likely to be sequencing artifacts and removes redundant edges. Finally, the layout algorithm finds unambiguous regions of the graph. After reads are ordered and oriented, an alignment is constructed from overlaps, and a consensus sequence is inferred. The overlap computation has a particular bottleneck. Naive methods that scaled quadratically were impractical when faced with datasets containing hundreds of Gb of sequences. For these reasons, most works on assembling high-throughput short-read sequence data have relied on the de Bruijn graph approach.

b. de Bruijn graphs

The foundation for de Bruijn graph-based assembly of whole-genome sequencing is presented in [1]. Each read is broken into a sequence of overlapping *k-mers*. The distinct *k-mers* are added as vertices to the graph, and *k-mers* that originate from adjacent positions in a read are linked by an edge. The assembly problem can be formulated as finding a walk through the graph that visits each edge in the graph once – a Eulerian path problem. In most instances, the assembler attempts to construct contigs consisting of unambiguous, unbranching regions of the graph.

The de Bruijn graph approach has a significant computational advantage when compared with overlap-based assembly strategies since it does not require finding overlaps between pairs of reads and, therefore, it does not require expensive dynamic programming procedures to identify such overlaps. Instead, the overlap between reads is implicit in the structure of the graph. The graph can easily be constructed as first *f k-mers* can be extracted from reads and added as vertices in the graph and second adjacent *k-mers* can be extracted from reads and added as edges. With suitable choices of data structures to represent the graph, this process can be completed nearly as fast as data can be read from disk. However, due to high memory costs of the de Bruijn assembly, it could be computationally infeasible for processing *e.g.*, mammalian genomes. There is approximately one *k-mer* for ever *y* base in a genome; then, de Bruijn graph of mammalian genomes has billions of vertices. ABySS assembler introduced a representation of the graph that did not explicitly store edges and SOAP *de novo* assembler uses a similar technique improving the memory consumption.

c. String graphs.

The de Bruijn graph has the elegant property that repeats get collapsed. All copies of a repeat are represented as a single segment in the graph with multiple entries and exit points. Myers [16] observed that a similar property could be obtained for overlap-based assembly methods by performing two transformations of an overlap graph. First, contained reads (substrings of other reads) are removed. Second, transitive edges are removed from the graph. The resulting string graph shares many properties with the de Bruijn graph without the need to break their ads into *k-mers*. The Edena assembler applied the string graph with early short-read sequencing data.

C. Modeling Mate-Pairs

Mate-pair information provides valuable constraints on the relative placement of sequence reads in an assembly. These constraints have been used to check the correctness of the assembly. Mate-pair information can also be used to link independent contigs into scaffolds-groups of contigs whose relative order and orientation are known and that are separated by gaps of approximately known size. The gaps between contigs represent sections of the genome that could not be reconstructed by the assembler owing to either missing data as the systematic bias of DNA sequencing or repeats. Mate-pair information can also be used to resolve certain repeats, potentially leading to longer and more accurate contigs. ALLPATHS assembler attempts to enumerate all paths connecting endpoints of a mate pair; to modify the de Bruijn graph to encode the mate-pair information; thereby resolving segments of graph consistent with the mate-pair information.

IV. Designing ArrOW

A. The ArrOW Conceptual Specification

Assembly experiments can be divided into three main macro-activities (Figure 1), each one decomposed into one or more activities of ArrOW. (I) *Macro-Activity Trimming Quality of Reads* executes the activity (1) Sickle. (II) *Macro-Activity Assembling Reads* executes activities (2) shuffle, (3) velveth, (4) velvetg (from Velvet), (5) MetaVelvet, and (6) Ray. (III) *Macro-Activity Calculating Length Distribution of Assemblies* executes activities (7) assemstat1, (8) assemstat2, and (9) assemstat3 (from assemstat).

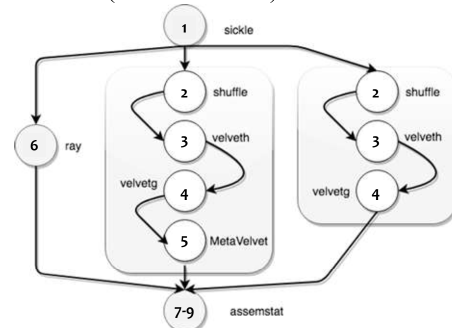

Figure 1. The conceptual view of the workflow ArrOW

The first activity executes the library sickle, which trims Illumina paired-end reads from 3’ to 5’ end using a sliding window technique (drops below 20 value) and generates as results two trimmed files for forward and reverse reads. After the execution of sickle, ArrOW invokes in parallel the assemblers Velvet (activities 2, 3, 4), MetaVelvet (activity 5), and Ray (activity 6) that use as inputs the trimmed files and generate assembled reads and contigs into a file.

Velvet is a de Bruijn assembler. It executes three steps: *shuffleSequences_fastq.pl* script converts reads to interleaved format file; *velveth* uses the information of each interleaved reads to construct the dataset (hashes reads) for *velvetg*; which builds the de Bruijn graph from *k-mers* (*k*=21), runs simplification and error correction over the graph, extracts contigs, and creates an output directory “out 21 velvet”. MetaVelvet is a modification and extension of the single-genome and de Bruijn-graph based assembler (Velvet), for short reads to metagenome assembly. It is executed on directory “out 21 velvet”. Ray improves on the standard de Bruijn graph by employing greedy heuristics; it executes the activity 6. Finally, the activities 7, 8, and 9 execute the in-

house script *assemstats.py*, for the three assemblers, and calculates the length distributions of the resulting assemblies.

B. Modeling ArrOW using SciCumulus

ArrOW could be executed by any parallel SWfMS. SciCumulus was our choice since it supports HPC and provenance features, as have been demonstrated in bioinformatics [17]. ArrOW activities are instrumented using the scripts of configuration (template *"xml"*) and execution (extractor *"sh"*) of SciCumulus. Instrumentation allows capturing and storing workflow metadata in the provenance database. The metadata and records information from assembly experiments – length distribution, total number of contigs or N50 contig length – can be used with performance results of the workflow. There are several advantages to using SciCumulus that are not available in other SWfMSs. Scientists can query the provenance database at runtime to help in workflow configuration, reuse previously related workflows or model a new one. These queries can be as simple as "Obtain statistics of ArrOW executions" as presented in Query 1 and Figure 2(A) or "Retrieve names, sizes and locations of ArrOW data files with the extension '.unpaired.fastq'" as presented in Query 2; or more complex by mixing these simple ones *e.g.,* extraction of domain-specific data stored in data files.

The second benefit is related to the scheduling cost model of SciCumulus. Since ArrOW activities have heterogeneous execution time distribution, SciCumulus can schedule short-term activities to less powerful virtual machines (VMs) and long-term activities to more powerful VMs and to scale the amount of VMs up and down according to performance behavior. For example, Ray is computing intensive and demands more capacity power (CPU, memory) than other assemblers. By querying Ray execution history in the provenance database, SciCumulus scales up the amount of VMs to improve the performance. The third benefit is related to the fault tolerance. Each execution of ArrOW has about 6.1% of activity execution failures. These faulty executions have to be aborted and SciCumulus has to restart each activity. Since it has all the information stored in the provenance repository it does not need to restart the entire workflow. It is easy to find and re-execute only failed activities.

V. EXPERIMENTAL EVALUATION

A. Environment Setup

SciCumulus is based on an algebraic approach where each activity receives a relation (table of several independent tuples) as input and independently processes each tuple. By storing the provenance data of workflow executions, powerful domain-specific queries can be defined. For example, for each parameter, SciCumulus records all steps and files associated with executed activities with this parameter in the provenance database. Records are queried for systematic analysess of the experiment in partial, or as a whole, after its completion. We have deployed SciCumulus on top of Amazon AWS. Amazon AWS is one of the most popular cloud providers, and many scientific and commercial applications are being deployed on it. It provides several different types of VMs with unique characteristics (CPU, RAM, storage capacity). In this paper, Amazon's C5 types have just considered: c5.9xlarge (36 cores, 72GB RAM), c5.4xlarge (16 cores, 32GB RAM), and c5.xlarge (4 cores, 8GB RAM). Each VM uses Linux Cent OS (64-bit) and was configured with necessary software and libraries. VMs can be accessed using SSH without password checking (although this is not recommended due to security issues). In terms of software, all VMs, no matter its type, execute the same programs and configurations. According to Amazon AWS, all VMs were instantiated in the US East-N. Virginia location and follow the pricing rules of that locality.

B. Experiment Setup

The 1,000 metagenomics raw datasets were downloaded from the NCBI Assembly, which are composed of different sizes of sequence reads. For instance the human metagenome anterior nares assembly data sequenced by Illumina (paired-end reads) has a GenBank entry (ID) with *Assembly Name* SRS018585 and *Assembly Accession* GCA_900218245. Other biological metadata are project definition, accession reference, the link of the FASTA sequence, etc. The dataset SRS018585 is composed of three files (SRS018585.trimmed.)*1.fastq, *.2.fastq, and *.singleton.fastq.

To evidence the advantages of the SciCumulus adaptation with respect to size of sequence files, we fixed the assembly program *i.e.,* independently of sequence size we processed the entire dataset with the three assemblers. The assembly software were configured with default parameters: Sickle 1.3, Velvet 1.2.08, MetaVelvet 1.2.02, Ray 2.2.0, and assemstat script. According to Khan *et al.*, [22], there is a clear association between assembly constructions of Ray, Velvet, and MetaVelvet, which was observed by our results. The comparative study of seven assemblers (ABySS, Velvet, Edena, SGA, Ray, SSAKE, Perga) were used datasets from the Illumina platform. Results showed that Velvet and ABySS outperformed in all assemblers with low assembling time and high accuracy values (number of contigs, high N50 length). Velvet consumed the least amount of memory and Ray was extremely high time computing. This type of study provides assistance to the scientists for selecting the suitable assembler according to the data and computational environmental.

C. Performance Evaluation of ArrOW

Before discussing the overall performance of ArrOW, we analyze the execute time of each workflow activity. By querying the provenance repository of SciCumulus, the statistics related to the execution time of all ArrOW activities are extracted as shown in Figure 2. Figure 2(A) presents the statistics values of ArrOW's activity executions of average execution time (770.16 sec.), median (14.0 sec.), and maximum (2,660.0 sec.). The main advantage of such distribution is to demonstrate that SciCumulus is able to distribute compute-intensive (*i.e.,* long-term) executions to more powerful VMs. On the other hand, SciCumulus dispatches less intensive (short-term) executions to less power VMs. It is worth mentioning that most activities of ArrOW are short-term ones. The activities that present the long-term executions are associated with MetaVelvet and Ray. Figure 2(B) presents the distribution of the total execution time (*i.e.,* TET) *per* activity considering a 16 cores execution.

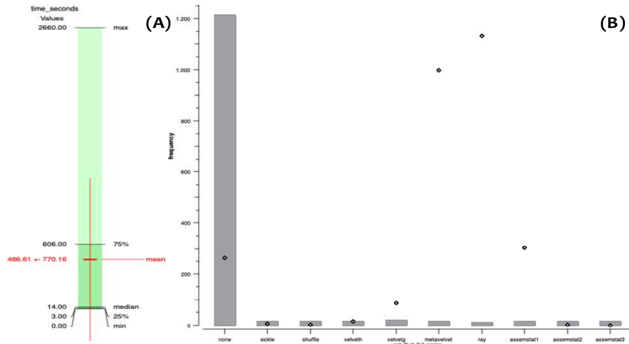

Figure 2. (A) The distribution of the execution time statistics of ArrOW. (B) The execution time histogram of ArrOW per activity

We first measure the performance of all programs (Figure 3) on a single VM to analyze the local optimization before adding more VMs to the execution pool. We measured the scalability of ArrOW using a combination of c5.9xlarge, c5.4xlarge, and c5.xlarge VMs up to 256 virtual cores. As the number of VMs increases (number of virtual cores), the TET of ArrOW executions decreases, as expected (Figure 3). The performance gains are very encouraging, *e.g.*, when ArrOW processes 1,000 assembly reads, the TET was reduced from 1.2 days (using 8 cores) to 3.4 hours (using 256 cores).

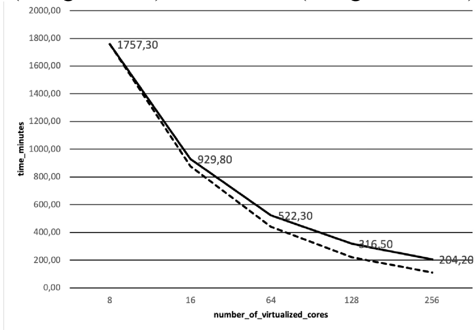

Figure 3. The total execution time of ArrOW

Even with cloud performance fluctuations, when using 16 cores ArrOW is approximately 14.55 times faster than the best-performing workflow execution on a single core. There is always a gain by adding more cores to the execution, from 8 up to 256 cores, but the efficiency (Figure 4) presents a degradation in all executions since VMs are heterogeneous and load balancing becomes more complex, thus introducing overhead in activity distributions by SciCumulus. Results indicate that acquiring more than 16 cores may not bring the expected benefit, particularly if financial costs are involved.

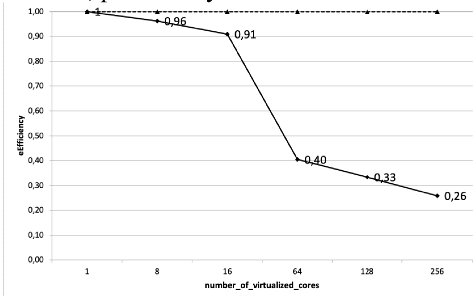

Figure 4. The efficiency of ArrOW

Overall, our performance analyses were eased by the information obtained by querying SciCumulus provenance repository after the workflow termination. The provenance repository of SciCumulus is based on the W3C PROV and PROV-Wf models. Due to that, the following query was executed to extract the desired information.

Query 1: "Obtain TET, statistical averages and biological information related to ArrOW executions". Querying provenance repository allows extracting the execution time for all executions of ArrOW activities. This information is very useful in large-scale experiments, as it is possible to know how workflows are running, or if any execution fails. It is possible that errors are related with size and origin of reads (genomes or metagenomes), or high *k-mer* values used in programs that cannot be processed by assembler.

```
SELECT a.tag,
        min(extract ('epoch' from (t.endtime-t.starttime))),
        max(extract ('epoch' from (t.endtime-t.starttime))),
        sum(extract ('epoch' from (t.endtime-t.starttime))),
        avg(extract ('epoch' from (t.endtime-t.starttime)))
FROM hworkflow w, hactivity a, hactivation t
WHERE w.wkfid = a.wkfid
AND a.actid = t.actid
AND w.wkfid =382
GROUP BY a.tag
```

Figure 2(B) shows results of Query 1 to evaluate the time required for the assembly processing for each read dataset, analyzing the influence in assembling execution time of input sizes, assemblers, and parameters. Moreover, TET (Figure 3) and efficiency (Figure 4) can be calculated by using Query 1.

D. *Biological Analysis*

Table 1 shows statistics of assembly processes using ArrOW with Velvet, MetaVelvet, and Ray. Outputs from assemblers were decomposed into contigs files used for analyses of accuracy of assemblers. The accuracy evaluation is based on total number of contigs and N50 contig length, which were collected with the Python script assemstats. In this analysis, *k-mer* was defined as 21.

TABLE 1. THE RESULTS OF ASSEMBLY PROCESSES FOR ARROW

Statistics / cut-off	*100*			*500*			*1,000*		
	Velvet	*MVelvet*	*Ray*	*Velvet*	*MVelvet*	*Ray*	*Velvet*	*MVelvet*	*Ray*
sum base	2892975	2889138	404510	63344	62287	2372	3802	4676	0
ncontig	30886	30884	2830	30883	30884	2830	30883	30884	0
cut_off	17913	17889	2830	100	96	4	3	4	0
min	100	100	100	500	500	515	1133	1026	0
med	147	147	128	565	579	564	1249	1088	0
mean	161	161	142	633	648	593	1267	1169	0
max	1420	1420	667	1420	1420	667	1420	1420	0
n50	6339	6330	1083	41	39	2	2	2	0
n50_len	165	165	139	618	646	626	1249	1142	0

In an ideal condition, the minimum number of contigs that matches the whole genome sequence could be generated from each assembly procedure. Scientists try to find out what each of the stats represents by varying the cut-off at 100, 500, 1,000, etc. At 1,450 of cut-off, all assemblers did not find sequences longer than the threshold. Results showed that on paired-end datasets, Ray assembled short reads into a relatively low number of contigs at 100 of cut-off. But at 1,000 Ray did not find sequences longer than the threshold. On paired-end datasets, Velvet produced high N50 contig length, whereas Ray produced low N50 contig length. One of the most often used statistics in assembly length distribution

comparisons is N50 length, a weighted median, where we weight each contig by the length. Fifty percent of bases in the assembly is contained in contigs shorter or equal to N50 length. Fifty percent of all bases in assembly is contained in contigs shorter or equal to N50 length. N50 is a statistical measure of average length of a set of sequences used widely in genomics.

Our results reinforce a previous comparative assembly study of Khan *et al*., [18] about a clear association of statistics obtained from assemblers as well as that Ray presents high CPU-memory requirements. It was also observed that Velvet is more scalable than Ray at running larger read datasets. In practice, an assembler which produces the fewer number of contigs with high N50 is considered as the ideal. We highlight that only our study in assembly experiments was designed for HPC and SWfMS based on provenance.

Query 2 extracts some biological results contained in the assembling outputs files '**ontigs.fa**'. **Query 2**: "Retrieve names, sizes, and locations of files with extension '**ontigs.fa**' produced by ArrOW and which workflow and activities produced those files". Query 2 is crucial for runtime monitoring of ArrOW workflow execution. It helps biological analysis at runtimeand to verify resulting length distributions obtained after the assembly (contained in '**ontigs.fa**' files).

Without querying the provenance database with Query 2, scientists would need to browse all directories manually and search which pairs were assembled successfully. Then they would need to separate and open these files to extract the information of the assembly process. The provenance database stores all this data and its relationships on a structured model. Thus it simplifies the querying process and allows for long-term analyses over experimental data.

VI. Final Remarks and Future Work

Assembly workflows executed by SWfMS can manage comparisons of a large volume of datasets and parameters variability as the *k-mer* values, reducing the long processing time for assembly analyses. In this paper, we proposed the ArrOW workflow to execute and manage assembly data-intensive experiments aiming the evaluation of three *de novo* sequence assemblers in terms of time execution, efficiency, and accuracy. ArrOW was executed with SciCumulus in Amazon AWS using parallel processing.

Our experiment evaluated 1,000 read sequence datasets. We found that each assembler is capable of assembling the whole genome but Ray assembler is the most time-expensive. Khan *et al*., [18] reported that Ray is not capable of assembling a eukaryotic genome with an environment about 4 GB of RAM or less. Velvet produced generally the best results among all three assemblers with comparatively low assembling time. The hybrid approach, Ray, also showed high N50 value, considered being ideal; however, the extremely high assembling time used by the Ray might make it prohibitively slow on large datasets.

ArrOW generated 9,000 workflow activity executions (1,000 executions for each of the 9 activities), producing 350 Gigabytes of data for the workflow execution. By analyzing the overall performance, through the provenance database, we state that ArrOW obtained significant gains with Velvet, MetaVelvet, and Ray. For example, executions with 256 cores reach performance improvements up to 88.3% for ArrOW. Based on the TET, speedup, efficiency, total assembly time (activities of assemblers Velvet, MetaVelvet, and Ray), total number of contig, and N50 contig length values, we observe that ArrOW with Velvet outperforms the others assemblers. Overall, the overhead imposed by the executions of ArrOW with SciCumulus is compensated by the advantages of data parallelism without too much effort from bioinformaticians. ArrOW results provide evidence that large computations involving assembly experiments can benefit from SciCumulus in clouds. Finally, the results presented in this paper can be extrapolated to the development of workflows in other areas that also require the exploration of large amounts of NGS data. As future work, we plan to explore complete human metagenomes of actual interest and to search for new candidate drug target enzymes.

Acknowledgment

The funding was provided by Brazilian agencies with the projects CNPq/Universal (Grant no. 429328/2016-8) and FAPERJ/JCNE (Grant no. 232985/2017-03). We are also grateful to the comments made by the anonymous referees.

References

[1] P. Compeau, P. Pevzner, and G. Tesler, "How to apply de Bruijn graphs to genome assembly," *Nat Biotech*, vol. 29, no. 11, 987–991 2011.
[2] J. Bang-Jensen, G. Gutin, and A. Yeo, "When the greedy algorithm fails," *Disc Opt*, vol. 1, no. 2, 121–127 2004.
[3] A. Tritt, "An Integrated Pipeline for de Novo Assembly of Microbial Genomes," *PLoS ONE*, vol. 7, no. 9, p. e42304. 2012.
[4] D. Mapleson, N. Drou, and D. Swarbreck, "RAMPART: a workflow management system for de novo genome assembly," *Bioinformatics*, vol. 31, no. 11, 1824–1826. 2015.
[5] E. M. Robertsen *et al.*, "META-pipe - Pipeline Annotation, Analysis and Visualization of Marine Metagenomic Data," *arXiv:1604*. 2016.
[6] D. Oliveira, F. A. Baião, and M. Mattoso, "Towards a Taxonomy for Cloud Computing from an e-Science Perspective," in *Cloud Comp*, N. Antonopoulos and L. Gillam, Eds. London: Springer, 2010, 47–62.
[7] J. Freire, "Provenance for Computational Tasks: A Survey," *Computing in Science Engineering*, vol. 10, no. 3, 11–21, 2008.
[8] M. Abouelhoda, S. Issa, and M. Ghanem, "Tavaxy: Integrating Taverna and Galaxy workflows with cloud computing," *BMC Bioinf.*, vol. 13, p. 77, 2012.
[9] "Amazon Elastic Compute Cloud," http://aws.amazon.com/ec2/.
[10] S. Koren, "Automated ensemble assembly and validation of microbial genomes," *BMC Bioinformatics*, vol. 15, p. 1262014.
[11] D. Oliveira, "SciCumulus: A Lightweight Cloud Middleware to Explore Many Task Computing Paradigm in Scientific Workflows," in *Int Conf Cloud Comp*, Washington, USA, 2010, 378–385.
[12] S. El-Metwally, "Next-Generation Sequence Assembly: Four Stages of Data Processing and Computational Challenges," *PLoS Comp Biol*, vol. 9, no. 12, p. e5. 2013.
[13] R. Ekblom and J. Wolf, "A field guide to whole-genome sequencing, assembly and annotation," *Evol App*, vol. 7, no. 9, 1026–1042. 2014.
[14] Y. Wan, "VirAmp: a galaxy-based viral genome assembly pipeline," *GigaScience*, vol. 4, no. 1. 2015.
[15] E. S. Lander and M Waterman, "Genomic mapping by fingerprinting random clones: a mathematical analysis," *Genomics*, vol. 2, no. 3, 231–239. 1988.
[16] E. Myers, "The fragment assembly string graph," vol. 21 Sup 2, 79–85. 2005.
[17] K. Ocaña and D. Oliveira, "Parallel computing in genomic research: advances and applications," *Adv App Chem*, p. 23. 2015.
[18] A. R. Khan, "A Comprehensive Study of De Novo Genome Assemblers: Current Challenges and Future Prospective," *Evol. Bioinform. Online*, vol. 14, p. 50, 2018.