SORA: Scalable Overlap-graph Reduction Algorithms for Genome Assembly using Apache Spark in the Cloud

Alexander J. Paul[1], Dylan Lawrence[2], Myoungkyu Song[3], Seung-Hwan Lim[4], Chongle Pan[5], Tae-Hyuk Ahn[6,*]
[1]Bioinformatics and Computational Biology Program, Saint Louis University, St. Louis, MO, US
[2]Computational and Systems Biology Program, Washington University in St. Louis, St. Louis, MO, US
[3]Department of Computer Science, University of Nebraska at Omaha, Omaha, NE, US
[4]Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, US
[5]School of Computer Science, University of Oklahoma, Norman, OK, US
[6]Department of Computer Science, Saint Louis University, St. Louis, MO, US
[1,6]{alex.paul,ted.ahn}@slu.edu,[2]dylan.lawrence@wustl.edu,[3]myoungkyu@unomaha.edu,[4]lims1@ornl.gov,[5]cpan@ou.edu
[*]Corresponding author

***Abstract*—The advent of high-throughput DNA sequencing techniques has permitted very high quality *de novo* assemblies of genomes, but raise an issue of requiring large amounts of computer memory to resolve the large genome graphs generated by most overlap graph *de novo* assemblers. To address these limitations, we present a novel algorithmic approach; Scalable Overlap-graph Reduction Algorithms (SORA). SORA adapts string graph reduction algorithms for the genome assembly using a distributed computing platform. To efficiently compute coverage for enormous paths in the graphs, SORA uses Apache Spark which is a cluster-based engine designed on top of Hadoop to handle very large datasets in the cloud. The experimental results show that SORA can process a nearly one billion edge graph in a distributed cloud cluster as well as smaller graphs on a local cluster with a short turnaround time. Moreover, our algorithms scale almost linearly with increasing numbers of virtual instances in the cloud. SORA is freely available for download at https://github.com/BioHPC/SORA/ .**

***Index Terms*—graph reduction, apache spark, genome assembly, cloud, overlap-layout-consensus,**

I. Introduction

De novo genome assembly—the reconstruction process of aligning and merging complete genome sequences from fragmented DNA sequences—is essential and inherently challenging in bioinformatics research [1]. After next-generation sequencing (NGS) techniques [2]–[4] have been introduced, tremendous changes and impacts happened on biological sciences, especially genomics [4]. NGS has revolutionized DNA sequencing by reducing the overall sequencing cost and increasing throughput, read length, and read accuracy; however, NGS requires analysis of a significant amount of sequencing data. According to previous studies [5], NGS has several limitations. For example, the fragmented DNA sequences (i.e., *reads*) have lengths (100∼400bp) which are shorter than the sequence lengths generated by first-generation Sanger sequencing (500∼1000bp). Third-generation sequencing techniques including Pacific Biosciences (PacBio) and Oxford Nanopore can provide long reads up to 60 kbp that greatly benefits in *de novo* genome assembly [6], but NGS is still dominant with low cost and low error rate.

The complexity of genome assembly is caused by multiple circumstances including the ploidy and heterozygosity, mainly affected by two factors: the number of reads and the length of the reads. Due to the complexity in genomic analysis, a time complexity and a space complexity of *de novo* assemblers are very sensitive for newly released sequencing techniques. To cope with both speed and sensitivity, most *de novo* genome assemblers adapt two widely used assembly paradigms: Overlap-layout-consensus (OLC) and de Bruijin graph (DBG) [7]. In the era of first-generation sequencing techniques, OLC approaches, such as Celera [8], have achieved adequate accuracy to accommodate the long reads and low sequencing depth generated by Sanger sequencing. Newbler [9], an assembler for Roche / 454 Life Sciences, also used OLC paradigm. Most OLC-based genome assemblers compute the sequence assembly of whole, complex genomes based on three steps: (1) finding **O**verlaps among all reads or between fragments to model a graph, (2) building a stretched **L**ayout of the overlap-graph, and (3) determining the most likely **C**onsensus sequence.

As an alternative approach to designing alignment and assembly algorithms, NGS techniques have emerged to produce shorter reads, but at much higher throughput. In NGS, DBG-based assemblers have been widely used to decompose reads into k-mers. Several techniques use memory-efficient DBG traversal to reduce the memory footprint of an assembly (e.g., efficient search for redundant k-mers), including Velvet [10], AbySS [11] and SOAPdenovo [12]. In contrast to the OLC approach which is less computationally efficient (e.g., expensive memory consumption and execution time for each assembler), most DBG assemblers use a genome-sized graph with less dependence on the sequencing depth, but create a larger memory overhead. The DBG approach performs relatively fast

978-1-5386-5488-0/18/$31.00 ©2018 IEEE

overlapping computation for short and high-throughput reads, while the OLC approach performs better for longer reads. These techniques use hashing based algorithms which have higher relative error rates but perform faster than the OLC approach. [13].

Recently, probabilistic algorithms using the MinHash technique have been developed to efficiently detect numerous overlaps between noisy, long reads from third-generation sequencing data [14], [15]. Canu, a successor of Celera assembler that is specifically designed for long and noisy single-molecule sequences [15]. However, assembling raw or processed sequences still generates a computationally expensive overlap graph that must be reduced or simplified. Additionally, the analysis of reads with a large amount of overlap is not easily parallelized as it requires extensive computational resources in both memory and processing time. As a result, these assemblers do not scale to large genomes. Few MPI-based scalable assemblers were previously proposed including Ray [16], AbySS [11], and SWAP2-Assembler [17], but the scalability and performance of the assemblers are limited and restricted.

To address these limitations, we present a novel OLC based algorithmic approach for genome assembly, called **S**calable **O**verlap-graph **R**eduction **A**lgorithms (SORA). SORA supports large-scale parallel processing by leveraging Apache Spark.[1] Spark provides an open source, general purpose, and distributed computing engine for cluster based computation [18], [19]. It performs efficient in-memory computation for fast large scale data processing in data intensive cluster computation. Unlike conventional distributed processing frameworks such as Apache Hadoop,[2] Spark caches datasets to memory which can increase the computational performance up to 100x faster for interactive jobs and iterative analytics.

Based on the Spark's cluster computing engine, SORA computes the genome assemblies by resolving repetitive sequences either in the cloud or on a local cluster system. SORA performs genome assembly by applying three overlap-graph reduction algorithms, *Transitive Edge Reduction*, *Dead-End Removal*, and *Composite Edge Contraction* algorithms, respectively. It processes a large-scale dataset containing a graph with nearly one billion edges with a short turnaround time on a distributed cloud computing cluster as well as a smaller dataset containing a graph with over 8 million edges on a local computing cluster. Spaler [20] is another Apache Spark and GraphX based *de novo* genome assembler using DBG construction and contraction, but SORA is the first proposed Spark-based scalable assembler using OLC approach in our limited knowledge. Our benchmark results demonstrate two main thrusts; (1) SORA realizes a cloud scalable *de novo* genome assembler by leveraging a state of the art graph processing framework, Spark; (2) SORA shows the applicability of cloud computing infrastructures to genome assembly and other biological applications using graph algorithms.

[1]Apache Spark – https://spark.apache.org

[2]Apache Hadoop – http://hadoop.apache.org

II. Background

A. Overlap-Layout-Consensus

The first step in OLC, Overlap, typically finds overlaps of reads by all-to-all pair-wise alignments. To find overlaps between reads efficiently a prefix/suffix hash table technique is a popular technique for overlap-based genome assembly [21]. To efficiently find all reads overlapping with a read *r*, every proper substring of minimum overlap in read *r* is searched in the hash table, and all retrieved reads are compared with the read *r*. As a result, the Overlap step constructs an overlap-graph that places reads as nodes and assigns an edge between two nodes when the two corresponding reads overlap by more than a specified cutoff. The number of nodes is equal to the number of unique reads, but the number of edges is dependent on the number of overlaps between reads. In the Layout and Consensus steps, the constructed overlap-graph is reduced and stretched into the most probable contiguous sequences, called contigs. The Layout step is a Hamiltonian path problem that is required to travel every read in the graph to produce longer sequences. The final Consensus step considers the alignment of all the original reads onto the draft contigs from the Layout step and utilizes a simple majority-based consensus to clean up draft sequences. To decrease extraneous edges in the graph, SORA adapts three overlap-graph reduction algorithms such as Transitive Edge Reduction (TER), Composite Edge Contraction (CEC), and Dead-End Removal (DER) [22]. Section III describes the details of how SORA applies these overlap-graph reduction algorithms.

B. Apache Spark

Apache Spark is a cluster-based engine to process large-scale datasets. In contrast to Hadoop that processes data on-disk, Spark provides a built-in batching system which handles input data stream in-memory, divides the data into batches for each node of a cluster, and generates the final stream of results in batches. To support a distributed graph-parallel computation in Spark, GraphX supplies graph abstraction models and a set of fundamental operations. It allows SORA to manipulate and execute queries on graphs represented as database entries. The design and implementation in SORA leverages a collection of computational operations in GraphX for graph loading, construction, transformation, and computations.

Although SORA is currently implemented in Scala, the design is portable and the core components are designed to be adapted with lower development costs by using other programming languages such as Java or Python.

III. Overlap-Graph Reduction Algorithms

This section describes how SORA adapts three overlap-graph reduction algorithms to the distributed cloud computing cluster using Spark. Figure 1 illustrates the overview of each workflow how each algorithm computes the overlap-graph reduction.

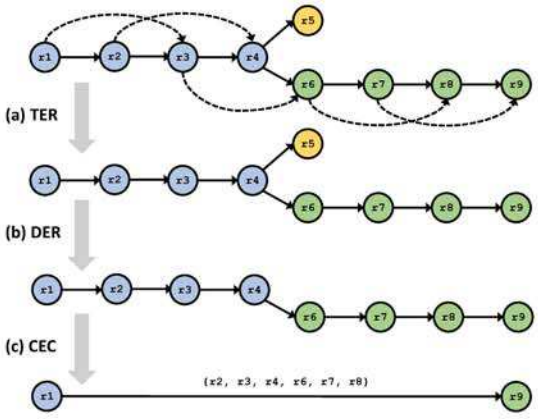

Fig. 1. The overlap-graph reduction algorithms: (a) Transitive Edge Reduction (TER), (b) Dead-End Removal (DER), and (c) Composite Edge Contraction (CEC).

A. Transitive Edge Reduction

Transitive edge reduction is a method of reducing complexity in graphs and helps provide clearer contigs by eliminating extraneous paths in the graph. After finding overlaps, the initial overlap graph contains many unnecessary edges. Note that read a overlaps read b which overlaps read c subsequently, which results in a shorter overlap length between read a and read c. Then, the string graph edge $a \rightarrow c$ is unnecessary because one can use the edges $a \rightarrow b \rightarrow c$ without $a \rightarrow c$ to spell the same sequence. The edge $a \rightarrow c$ is then identified as a transitive edge and is deleted. Removing all transitive edges significantly simplifies the overlap graph without losing any information.

The general transitive edge reduction algorithm takes $O(ED)$ time where E is the number of edges and D is the maximum out degree for the read, but Myer proposed a linear $O(E)$ expected time transitive reduction algorithm shown in [22]. In Algorithm 1 we use the GraphX library operators to implement the transitive edge reduction algorithm based on graph-parallel abstraction. After constructing the initial property graph from the edge properties, the `aggregateMessages` operator can compute the set of neighbors for each vertex and retrieve the edge properties including overlap length at the same time. The required set of neighbors can be joined with the graph using `outerJoinVertices`. After comparing overlap lengths of the edges for each vertex in parallel, the edges are marked as `TRUE` if the edges can be removed. The `subgraph` operator returns a new graph containing only the edges not marked for removal.

B. Dead-End Removal

Dead-End Removal (DER) eliminates short dead-ends or spurs from the graph, reduces erroneous reads, and decreases the graph complexity. The short dead-end paths are mostly caused by sequencing errors and false-positive joins of overlapping of chimeric sequences. Most assemblers identify the dead-ends by considering short length edges with low-depth coverage to be dead-ends.

Algorithm 1 Spark Algorithm for Transitive Edge Reduction

Input: Let overlapG be an overlap graph G(V, E).
Output: Let reducedG be a reduced graph G(V', E').

```
// Compute the set of neighbors for each vertex.
neighborVtx = aggregateMessages(overlapG(V, E)) {
    for v ∈ V do
        ort ← getOrientation(v)
        sendToSrc(getDstId(v), ort, getOverlapLen(v))
        sendToDst(getSrcId(v), ort, getOverlapLen(v))
    end for
}
// Join graph with neighbors.
joinedG = outerJoinVertices(overlapG(V,E),neighborVtx)
// Traverse each edge and mark true if the edge is removable.
markedG = mapTriplets(joinedG(V, E)) {
    for e ∈ edges of adjacent vertices of a vertex in V do
        if getOverlapLen(e) < getMaxOverlapLen(e) then
            e ← true
        end if
    end for
}
// Remove the marked edge using subgraph.
reducedG = subgraph(markedG(V, E))
```

Algorithm 2 Spark Algorithm for Dead End Removal

Input: Let overlapG be an overlap graph G(V, E).
Output: Let reducedG be a reduced graph G(V', E').

```
// Compute the in/out edge counts for each vertex.
inOutVtx = aggregateMessages(overlapG(V, E)) {
    for v ∈ V do
        ort ← getOrientation(v)
        if ort == ⟷ then
            sendToSrc(1,0); sendToDst(1,0)
        end if
        if ort == > −− < then
            sendToSrc(0,1); sendToDst(0,1)
        end if
        if ort == ↣ || ort == ↢ then
            sendToSrc(0,1); sendToDst(1,0)
        end if
    end for
}
// Join graph with neighbors.
joinedG = outerJoinVertices(overlapG(V, E), inOutVtx)
// Traverse each edge and mark true if the edge is removable.
markedG = mapTriplets(joinedG(V, E)) {
    for e ∈ edges of adjacent vertices of a vertex in V do
        if e.out == 0 then
            e ← true
        end if
    end for
}
// Remove the marked edge using subgraph.
reducedG = subgraph(markedG(V, E))
```

Algorithm 2 describes the DER algorithm based on the GraphX operators. Algorithm 2 takes as input the reduced graph that Algorithm 1 has produced as the output and executes the `aggregateMessages` operator to compute the number of edges going in and out of each vertex depending on the orientation of the edge. This information can be joined with the input reduced graph by using `outerJoinVertices`. In parallel, if the number of outgoing edges from a node is zero and the edge can be removed mark the edge `TRUE`. The `subgraph` operator returns a new graph with the edges marked `TRUE` removed.

C. Composite Edge Contraction

Composite Edge Contraction (CEC) reduces the computational complexity by processing larger volumes of data in the

graph. Especially, CEC merges vertices guaranteed to process the graph without loss of information. In the case of OLC, a read is represented for branching to two additional reads which deviate from each other by one nucleotide, both of which then overlap back to the same read. In contrast to OLC, the CEC algorithm simplifies the path analysis by removing redundancy and reducing complexity of the graph, considering only the contractible edges without loss of information. To simplify the overlap graph, a simple vertex, r, along with its in-arrow edge (u, r) and out-arrow edge (r,w), are replaced by a composite edge (u,w) in the overlap graph.

Algorithm 3 describes the CEC by using the operators of GraphX and GraphFrames. After receiving the reduced graph from Algorithm 2, the operator `aggregateMessages` computes the number of edges going in and out of each vertex depending on the orientation of the edge. The result of a processed set of vertices and edges is integrated with the input reduced graph by using the operator `outerJoinVertices`. The operator `mapTriplets` is parallelized to investigate the edges of each adjacent vertex to determine whether the vertex only includes a pair of incoming and outgoing edges. It then marks the edge `TRUE` if they can be contracted. The `subgraph` operator returns a new graph with only the contractable edges. The operator `connectedComponent` identifies the connection relationship among contractible vertices and produces the vertex information with the vertex IDs for the connected contractible subgraphs. Given the contractible vertex information, the operator `innerJoin` performs an inner join between each contractible and internal vertex to produce a set of the new vertex properties, which is used in the operator `aggregateUsingIndex` to aggregate the contracted vertices ensuring consistency by joining the IDs among vertices. Then, the operator `subgraph` filters out the edges marked `FALSE` to remove the contractible edges from the original graph. Finally, the operator `outerJoinVertices` generates the contracted edges, which parameterize the operator `graph` to construct a new reduced graph.

IV. Experimental Results

Figure 2 shows a practical pipeline of genome assembly using SORA. In our experiments, we applied three overlap-graph reduction algorithms (Transitive Edge Reduction, Dead-End Removal, and Composite Edge Contraction) in SORA to two different types of benchmark datasets. For the first experiment described in Section IV-A, we downloaded a metagenomic dataset from a repository Sequence Read Archive at the National Center for Biotechnology Information (NCBI) [21].[3] The metagenomic dataset is considerably large containing mixed DNA especially 64 diverse bacterial and archaeal microorganisms [23]. For the second experiment described in Section IV-B, we obtained a single genome dataset of *Conyza canadensis* (also known as horseweed) [24].

[3]The accession number is SRX200676.

Algorithm 3 Spark Algorithm for Composite Edge Contraction

Input: Let overlapG be an overlap graph G(V, E).
Output: Let contractedG be a contracted graph G(V', E').

```
// Compute the in/out edge counts for each vertex.
inOutVtx = aggregateMessages(overlapG(V, E)) {
    for v ∈ V do
        ort ← getOrientation(v)
        if ort == ⟷ then
            sendToSrc(1,0); sendToDst(1,0)
        end if
        if ort == > −− < then
            sendToSrc(0,1); sendToDst(0,1)
        end if
        if ort == ⤖ or ort == ⟻ then
            sendToSrc(0,1); sendToDst(1,0)
        end if
    end for
}
// Join graph vertices with in/out edge counts.
joinedG = outerJoinVertices(overlapG(V, E), inOutVtx)
// Traverse each edge and mark true if edge is contractable.
markedG = mapTriplets(joinedG(V, E)) {
    for e ∈ edges of adjacent vertices of a vertex in V do
        if e.out == 1 and e.in == 1 then
            e ← true
        end if
    end for
// Remove the edges marked true using subgraph.
contraG = subgraph(markedG(V, E))
// Calculate the connected components for each node.
conVtx = connectedComponents(contraG(V, E))
// Combine connected vertices with graph.
dupVtx = vertices.innerjoin(markedG(V, E), conVtx)
contraVtx = vertices.aggregateUsingIndex(markedG(V, E), dupVtx)
// Remove the edges marked false using subgraph.
remainedG = subgraph(markgedG(V, E))
contraEdges = outerJoinVertices(remainedG(V, E), conVtx)
// Generate a new graph using the modified edges and vertices.
contractedG = graph(contraVtx, contraEdges)
```

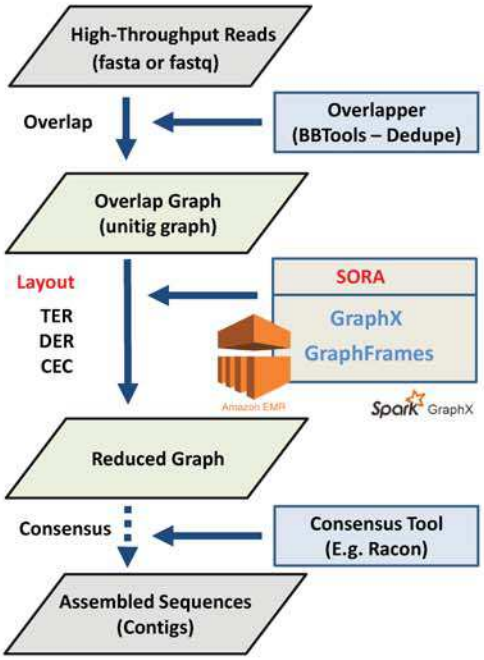

Fig. 2. Overview of analysis pipeline using SORA.

A. Metagenomic Dataset Analysis

In the experiment, we observed that SORA significantly reduced the number of reads in the metagenomic datasets,

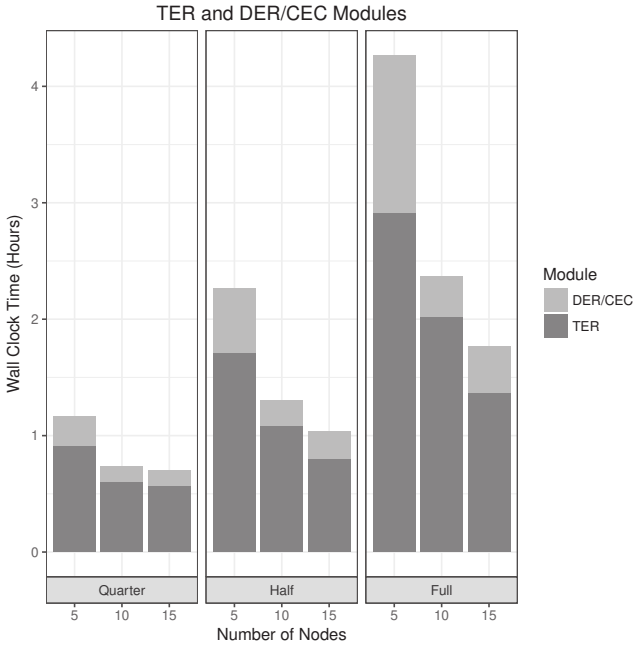

Fig. 3. Wall-clock time for different number of nodes with the metagenomic dataset.

which consequently allows binning of the contigs to reconstruct genomic bins more quickly and efficiently. The benchmark was performed on Amazon Web Service (AWS) Elastic Computing Cloud (EC2) 15 m4.xlarge instances where each instance (node) has 2.3 GHz Intel Xeon E5-2686 v4 (Broadwell) processors (4 vCPU) and 16 GB memory.

1) Overlap Graph Construction: The sequence dataset obtained from NCBI contains 100-bp roughly 109 million paired-end. Reads shorter than 60bp and containing multiple Ns were removed using Sickle (https://github.com/najoshi/sickle) and error correction was done using the BBNorm (https://sourceforge.net/projects/bbmap).

2) Transitive Edge Reduction: The Transitive Edge Reduction (TER) step showed a drastic reduction on the number of edges in the graph. Table I shows the reduction results according to three types of data size such as quarter, half, and full datasets. Given the full dataset with 868 million edges, the TER algorithm generated the reduced graph consisting of 57.4 million edges with 93.39% reduction.

Figure 3 shows the efficient scalability of the TER algorithm where the computational time decreases as the number of cluster nodes increases. For example, the TER algorithm completed the reduction of the graph using 5 cluster nodes in 2.92 hours, while completing with 15 cluster nodes in 1.37 hours.

3) Dead-End Removal and Composite Edge Contraction: Table I shows the evaluation results of the combination of two algorithms Dead-End Removal (DER) and Composite Edge Contraction (CEC) with the quarter, half, and full datasets. Given the quarter dataset containing 12.5 million edges, the combined DER-CEC algorithm generated the reduced graph

TABLE I
THE OVERLAP-GRAPH REDUCTION RESULTS WITH THE METAGENOMIC DATASET. #EDGE DENOTES THE NUMBER OF EDGES OF THE GRAPH AND TIME THE RUNNING TIME (HOURS) FOR THE COMPUTATION.

Algorithm	Size	#EDGE (before)	#EDGE (after)	TIME
TER	Quarter	217,002,504	12,482,946	0.57
	Half	434,005,009	23,818,401	0.80
	Full	868,010,019	57,363,515	1.37
DER-CEC	Quarter	12,482,946	469,130	0.13
	Half	23,818,401	763,474	0.23
	Full	57,363,515	2,341,610	0.40

with 0.5 million edges with 96% reduction; given the full dataset with 57.3 million edges, the DER-CEC algorithm resulted in the reduced graph comprising 2.3 million edges with 95.97% reduction.

Figure 3 shows the scalability of the combined DER-CEC algorithm by measuring each running time per different numbers of cluster nodes within the same sized dataset. As taking one of the best examples in the full dataset, we compared the running time between 5 and 15 cluster nodes. The DER-CEC algorithm completed the reduction of the graph using 5 cluster nodes in 1.35 hours, while fast completing with 15 cluster nodes in 0.4 hours.

4) Benchmark to Omega: To demonstrate the benefits of SORA's distributed cloud computation, we compared two approaches: SORA and Omega. Omega is another overlap graph metagenome assembler implemented in C++ [21]. We could choose another baseline application such as Spaler [20], which is a Spark-based *de novo* genome assembler using DBG, but it is not publicly available to download. SORA shows higher performance with 1.77 hour running time than Omega with 7.5 hour running time. In addition to efficient speedy performance, SORA uses the less amount of system memory compared to Omega since it breaks down the graph computation tasks to process them in parallel, thereby allowing more of the graph to be in memory and speeding up the analysis.

B. Horseweed Dataset Analysis

To show the flexibility and usability of SORA, we applied SORA to a single genome dataset to generate a reduced graph. Total size of 72 FASTQ paired-end files is 108 GB. We used a local computational workstation that has 32 cores (Intel Xeon Processor E5-2640 V3 2.6GHz) and 128 GB of memory (DDR4 2133MHz ECC).

1) Overlap Graph Construction: To demonstrate a large dataset genome assembly from single genome, we implemented a shell script as a batch process that performs error correction on the genome dataset, finds overlaps of the corrected reads and generates a large overlap graph, and thereafter executes SORA. The dataset was initially processed for the normalization and graph construction containing 8.3 million edges. The pipeline including SORA completed the assembly in 9.75 hours where the SORA core modules (TER, DER, and CEC) took less than 10 minutes.

2) Transitive Edge Reduction: Table II shows the evaluation result of the TER algorithm with the single genome dataset containing 8.3 million edges. The TER algorithm produced the reduced graph consisting of 5.4 million edges. This step also was efficient with memory consumption not requiring above 22% of overall memory (128 GB total system memory).

3) Dead-End Removal and Composite Edge Contraction: Table II also shows the results of overlap-graph reduction of the combined DER-CEC algorithm with the dataset—the graph containing 5.4 million edges that the TER algorithm generated above. The DER-CEC algorithm completed the computation in 8.23 minutes with the maximum 37% consumption of the 128 GB total memory.

TABLE II
THE OVERLAP-GRAPH REDUCTION RESULTS WITH THE HORSEWEED DATASET. #EDGE DENOTES THE NUMBER OF EDGES OF THE GRAPH AND TIME THE RUNNING TIME FOR THE COMPUTATION.

	#EDGE (before)	#EDGE (after)	TIME (mins)
TER	8,259,543	5,386,287	1.02
DER-CEC	5,386,287	1,027,959	8.23

V. DISCUSSION AND CONCLUSION

As the price of sequencing continues to drop and the emergence and fine tuning of new sequencing technologies increase the amount of raw data is growing exponentially. Current algorithms can handle the large influx of raw reads but require a large and expensive computational cluster with a large amount of computer memory. This is generally only available to large labs that can afford to buy and maintain a computational cluster. SORA helps bridge this gap for smaller labs by providing an efficient method for contigs generation using distributed computing in the cloud. This provides all labs the ability to analyze whole genome sequencing and generate novel sequenced contigs.

OLC models are a well-known method for *de novo* assembly, but can be problematic for large amounts of short reads leading to false alignments. This increases the computational memory needed for storing and analyzing the graphs generated from these reads efficiently. SORA can handle these problems by using the Apache Spark engine to manage the distributed computation in the cloud. Apache Spark efficiently uses in memory storage across multiple nodes to provide a performance boost compared to standard disk reading and writing which would normally be required for large datasets on a single computer.

ACKNOWLEDGEMENTS

TA is supported by NSF-1566292, NSF-1564894, Saint Louis University President's Research Fund 2018, and Amazon Web Service (AWS) Cloud Credits for Research. DL is supported by T32 HG000045 from the National Human Genome Research Institute.

REFERENCES

[1] P. Flicek and E. Birney, "Sense from sequence reads: methods for alignment and assembly," *Nat Methods*, vol. 6, no. 11 Suppl, pp. S6–S12.
[2] W. J. Ansorge, "Next-generation dna sequencing techniques," *New biotechnology*, vol. 25, no. 4, pp. 195–203, 2009.
[3] D. G. Hert, C. P. Fredlake, and A. E. Barron, "Advantages and limitations of next-generation sequencing technologies: a comparison of electrophoresis and non-electrophoresis methods," *Electrophoresis*, vol. 29, no. 23, pp. 4618–4626, 2008.
[4] M. L. Metzker, "Sequencing technologies - the next generation," *Nature Reviews Genetics*, vol. 11, no. 1, pp. 31–46.
[5] P. K. Wall, J. Leebens-Mack, A. S. Chanderbali, A. Barakat, E. Wolcott, H. Liang, L. Landherr, L. P. Tomsho, Y. Hu, J. E. Carlson *et al.*, "Comparison of next generation sequencing technologies for transcriptome characterization," *BMC genomics*, vol. 10, no. 1, p. 347, 2009.
[6] J. Eid and et al., "Real-time dna sequencing from single polymerase molecules," *Science*, vol. 323, no. 5910, pp. 133–8.
[7] J. R. Miller, S. Koren, and G. Sutton, "Assembly algorithms for next-generation sequencing data," *Genomics*, vol. 95, no. 6, pp. 315–27.
[8] E. W. Myers and et al., "A whole-genome assembly of drosophila," *Science*, vol. 287, no. 5461, pp. 2196–204.
[9] M. Margulies and et al., "Genome sequencing in microfabricated high-density picolitre reactors," *Nature*, vol. 437, no. 7057, pp. 376–80.
[10] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de bruijn graphs," *Genome Res*, vol. 18, no. 5, pp. 821–9.
[11] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "Abyss: a parallel assembler for short read sequence data," *Genome Res*, vol. 19, no. 6, pp. 1117–23.
[12] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, S. Li, H. Yang, J. Wang, and J. Wang, "De novo assembly of human genomes with massively parallel short read sequencing," *Genome Res*, vol. 20, no. 2, pp. 265–72.
[13] M. Pop, "Genome assembly reborn: recent computational challenges," *Brief Bioinform*, vol. 10, no. 4, pp. 354–66.
[14] K. Berlin, S. Koren, C. S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nat Biotechnol*, vol. 33, no. 6, pp. 623–30.
[15] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, "Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation," *Genome Res*, vol. 27, no. 5, pp. 722–736.
[16] S. Boisvert, F. Laviolette, and J. Corbeil, "Ray: Simultaneous assembly of reads from a mix of high-throughput sequencing technologies," *Journal of Computational Biology*, vol. 17, no. 11, pp. 1519–1533.
[17] J. Meng, S. Seo, P. Balaji, Y. Wei, B. Wang, and S. Feng, "Swap-assembler 2: Optimization of de novo genome assembler at extreme scale," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 195–204.
[18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," pp. 10–10, 2010.
[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," pp. 2–2, 2012.
[20] A. Abu-Doleh and U. V. Catalyurek, "Spaler: Spark and graphx based de novo genome assembler," in *2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 1013–1018.
[21] B. Haider, T.-H. Ahn, B. Bushnell, J. Chai, A. Copeland, and C. Pan, "Omega: an overlap-graph de novo assembler for metagenomics," *Bioinformatics*, vol. 30, no. 19, pp. 2717–22.
[22] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21 Suppl 2, pp. ii79–85.
[23] M. Shakya, C. Quince, J. H. Campbell, Z. K. Yang, C. W. Schadt, and M. Podar, "Comparative metagenomic and rrna microbial diversity characterization using archaeal and bacterial synthetic communities," *Environmental Microbiology*, vol. 15, no. 6, pp. 1882–1899.
[24] Y. Peng, Z. Lai, T. Lane, M. Nageswara-Rao, M. Okada, M. Jasieniuk, H. O'Geen, R. W. Kim, R. D. Sammons, L. H. Rieseberg, and C. N. Stewart, "De novo genome assembly of the economically important weed horseweed using integrated data from multiple sequencing platforms," vol. 166, no. 3, pp. 1241–1254, 2014.