Scalable De Novo Genome Assembly Using a Pregel-Like Graph-Parallel System

Guimu Guo, Hongzhi Chen, Da Yan, James Cheng, Jake Y. Chen, and Zechen Chong

Abstract—De novo genome assembly is the process of stitching short DNA sequences to generate longer DNA sequences, without using any reference sequence for alignment. It enables high-throughput genome sequencing and thus accelerates the discovery of new genomes. In this paper, we present a toolkit, called PPA-assembler, for de novo genome assembly in a distributed setting. The operations in our toolkit provide strong performance guarantees, and can be assembled to implement various sequencing strategies. PPA-assembler adopts the popular *de Bruijn graph* based approach for sequencing, and each operation is implemented as a program in Google's Pregel framework which can be easily deployed in a generic cluster. Experiments on large real and simulated datasets demonstrate that PPA-assembler is much more efficient than the state-of-the-arts while providing comparable sequencing quality. PPA-assembler has been open-sourced at https://github.com/yaobaiwei/PPA-Assembler.

Index Terms—Genome assembly, graph, distributed, vertex-centric, Pregel, DNA, read, contig, k-mer.

1 Introduction

Modern sequencing technologies generate a large number of short DNA segments called *reads*, which are stitched together to generate longer DNA sequences for finding new genomes. Although millions of reads can be generated in a day to allow high sequencing coverage, the assembly process is very costly. Single-threaded assemblers [16], [30], [24], [32], [2], [15], [33], [5], [7] often require a high-end server with terabytes of RAM, and are not efficient enough. As a result, parallel short read assembly has aroused a lot of attention recently thanks to the advances in big data systems. Many parallel (and often, distributed) assemblers have emerged, including ABySS [25], Spaler [1], Ray [3] and SWAP-Assembler [18]. However, they develop their respective distributed execution engines in an ad-hoc manner, without analyzing the quality and time cost guarantees.

To overcome this weakness, we developed a toolkit called PPA-assembler which implements the basic data structures and operations in de novo genome assembly. PPA-assembler decouples low-level execution (e.g., data distribution and communication) from the high-level assembly logic, allowing both layers to be independently optimized. The lower-level execution layer relies on Google's Pregel [17] framework which is optimized to deliver high execution throughput in a distributed cluster, and which also provides a user-friendly think-like-a-vertex programming interface to the upper-level algorithmic layer for ease of implementing and extending assembly strategies. Moreover, we have implemented common operations in de novo genome assembly with strong performance guarantees, and they can be assembled to implement various sequencing strategies. Further extensions are also made easy by the user-friendly Pregel model adopted.

We assume that readers are already familiar with the concepts in de Bruijn graph (abbr. DGB) based de novo genome assembly, such as reads, contigs, k-mers, reverse complement, directionality, tips and bubbles. If they are new to you, please first refer to Section III of our arXiv preprint [26] for a detailed tutorial.

Before presenting PPA-Assembler, we first list some examples of weaknesses caused by ad-hoc designs in existing assemblers:

- As the *Implementation* section of [25] indicates, ABySS needs to collect messages into larger 1KB packets for transmission in batch, in order to hide the round-trip time of individual messages; but this should be a detail in the communication layer rather than in algorithm design. Such communication details are automatically taken care of and optimized by a Pregel-like system.
- ABySS [25] builds a de Bruijn graph (DBG) by letting each k-mer send messages to its 8 possible neighbors (with A/T/G/C prepended/appended) to establish edges; but their method increases ambiguity and hence reduces contig length. For example, an edge will be created between 2-mers "CA" (e.g., contributed by 3-mer "CAT") and "AA" (e.g., contributed by "GAA") even though the 3-mer "CAA" may not exist in the DNA molecule.
- Spaler [1] iteratively breaks each unambiguous path in a DBG by sampled vertices to form segments, and then merges segments that meet at a sampled boundary vertex; this process is repeated until unambiguous nodes in the DBG account for more than 1/3 of all vertices in the graph. However, this heuristic provides no guarantee of path maximality, while as we shall see our PPA-assembler guarantees path maximality while providing a strict performance guarantee of logarithmic time complexity bound.

In fact, other than Spaler (built on top of Apache Spark), none of the other existing assemblers is Hadoop-compatible[1]. Hadoop compatibility is important for the existing Big Data ecosystem

- *Guimu Guo and Da Yan are with the Department of Computer Science, the University of Alabama at Birmingham.*
  Jake Y. Chen is with the Informatics Institute in School of Medicine, the University of Alabama at Birmingham.
  Zechen Chong is with the Department of Genetics and Informatics Institute at the University of Alabama at Birmingham.
  E-mails: {guimuguo, yanda, jakechen}@uab.edu, zchong@uabmc.edu
- *Hongzhi Chen and James Cheng are with the Chinese University of Hong Kong.*
  E-mails: {hzchen, jcheng}@cse.cuhk.edu.hk

1. Hadoop: http://hadoop.apache.org/

since it allows different tools to interoperate for completing a complex task. Genome assembly is especially so since it is just one operation in a genomic workflow: the input readers may come from a previous MapReduce job that prepares them with quality control, and the generated contigs may be used by subsequent analytic jobs running Spark MLlib. Different jobs usually exchange their data using Hadoop Distributed File System (HDFS).

Although Spaler [1] is Hadoop-compatible, the operations designed are rather ad hoc: they only demonstrate how genome assembly operations can be mapped into Spark API, without any formal analysis on the computation complexity. Moreover, most operations in DBG-based sequencing are graph operations, for which Spaler [1] uses the GraphX [9] (Spark's graph API) that are often over one order of magnitude slower than tailer-made Pregel-like systems [29], [4].

Our PPA-Assembler is naturally Hadoop-compatible and can be easily deployed in a generic cluster, since it is built on top of Google's Pregel framework, whose open-source Hadoop-compatible implementations are abundant including Pregel+ [27], Giraph [6], GraphX [9] and GPS [20]. PPA-assembler adopts the popular *de Bruijn graph* (DBG) based approach for sequencing [19], and we built it on top of Pregel+[2], our open-source implementation of Google's Pregel framework for big graph processing. However, we remark that our proposed algorithms are platform independent and can be implemented in any Pregel-like systems, and Pregel+ is chosen due to its superior performance as reported by [14] and its wide application [22], [8], [28], [27].

Since the assembly process also involves some non-graph operations, such as to construct DBG from raw DNA reads, we also extended Pregel+'s graph-parallel API with new functionalities, including grouping and merging data by key, and in-memory data conversion for seamless job concatenation. Each operation in PPA-Assembler is a Pregel+ program that may either read its input from HDFS, or directly obtain its input by converting the output of another operation in memory. This allows data to be carried over between different jobs without the need to being dumped to HDFS for loading back, and it shares a similar idea as the Resilient Distributed Dataset (RDD) of Spark [31].

The contributions of this work are summarized as follows:

- PPA-Assembler is built on top of Google's Pregel framework to be Hadoop-compatible, and avoids ad-hoc engine designs that are exposed to various weaknesses.
- Key operations in de novo genome assembly are implemented as Pregel programs, such as contig merging, tip removing and bubble filtering. Each operation is implemented as a *Practical Pregel Algorithm* (PPA) as defined in [28], which runs for at most logarithmic number of iterations (to DBG size), and each iteration has linear space usage, computation cost and communication cost.
- The key operations can be assembled to implement various assembly strategies, and the user-friendly Pregel API also makes it easy to develop new operations to extend the existing assembly workflow.
- Extensive experiments are conducted on real datasets from different species with different read length and depth, which shows that PPA-assembler is always much faster than other assemblers while achieving comparable sequencing quality.

2. Pregel+: http://www.cse.cuhk.edu.hk/pregelplus/

The rest of this paper is organized as follows. Section 2 first reports extensive experiments on performance comparison of PPA-assembler with existing state-of-the-art assemblers to demonstrate the superior performance of PPA-assembler. Section 3 reviews the framework of Pregel, and the definition of PPA. Section 4 presents the implementation of our various operations in PPA-assembler. Finally, we report additional experimental results regarding PPA-Assembler in Section 5, and conclude this paper in Section 6.

2 System Performance Comparison

As we have discussed, existing state-of-the-art parallel assemblers often use ad-hoc designs for their execution engines, leading to inferior performance and sometimes compromised sequencing quality. In contrast, built on top of Pregel with algorithmic performance guarantees, PPA-assembler is consistently much faster and delivers comparable sequencing quality as we shall show in this section, even though we are not implementing new operations in DBG-based genome assembly other than the standard ones (probably with even less operations than some existing assemblers). We, however, remark that PPA-assembler can be easily extended with new operations to improve sequencing quality further.

PPA-assembler has been open-sourced at https://github.com/yaobaiwei/PPA-Assembler, and this section reports its performance while comparing with the state-of-the-art assemblers.

Systems. For the purpose of comparison, we consider the state-of-the-art parallel assemblers ABySS [25] (version 2.1.5), Ray [3] (version 2.3.1) and SWAP-Assembler [18] (version 2); Spaler is not open-sourced and is thus not included in our comparison. We also consider the state-of-the-art single-machine assemblers Velvet [32] (version 1.2.10), SPAdes [2] (version 3.13.0) and SOAPdenovo2 [15]. Besides Ray, the other five assemblers are all DBG-based. In contrast, Ray uses subsequences called seeds which are heuristically extended to generate contigs.

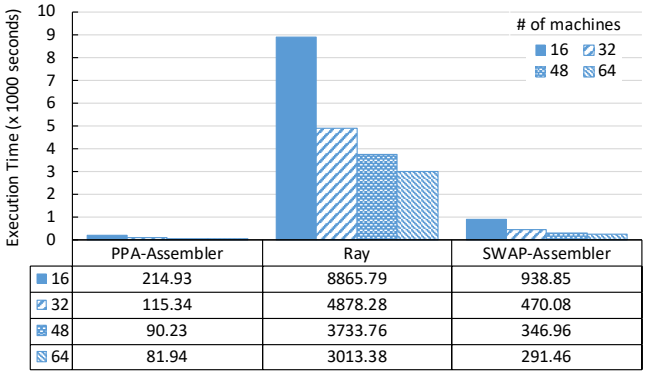
For our PPA-assembler, we adopt the simple workflow ①②③④⑤⑥②③ shown in Figure 17 (to be detailed in Section 4), i.e., to grow contigs once further after error correction. However, we remark that users may customize their own workflow or even change the existing operations (e.g., add coverage-threshold pruning to bubble filtering) or add new operations implemented in Pregel+'s API (e.g., branch splitting [1] for error correction) to implement different assembly strategies.

Settings. All experiments were conducted on a cluster of 15 machines connected by Gigabit Ethernet, each with 24GB memory, 8TB disk space, and 6 CPU cores (Intel Xeon X5650 @ 2.67GHz) with hyperthreading. We used $k = 31$ for defining k-mers as is commonly used. For PPA-assembler, we adopt the following parameters that are found to work well on various datasets; the sequencing results are very stable near these parameter ranges: (1) we filter any k-mers whose coverage by reads is below 5, (2) we set the length threshold for tip removal as 80, (3) we set the edit distance threshold for bubble filtering as 5 (i.e., two sequences are considered for bubble filtering only if they are within distance 5). For the other systems, we adopt their default settings.

Datasets. In order to demonstrate that the performance advantage of PPA-assembler is robust enough in different experimental conditions, we use diverse datasets of different read depth, different read length, and various species, as summarized in Table 1. All the datasets are in FASTQ format.

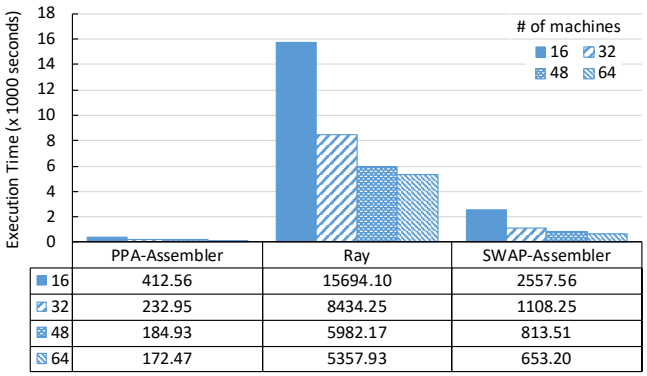
TABLE 1
Datasets (bp = base pairs)

Dataset Name	# of Reads	AVG Read Length	Reference Sequence Length	Read Coverage
Homo Sapiens Chromosome X (HCX-C6)	9,293,648	100 bp	158,270,121	6
Homo Sapiens Chromosome X (HCX-C10)	15,488,736	100 bp	158,270,121	10
Homo Sapiens Chromosome X (HCX-C20)	30,977,802	100 bp	158,270,121	20
Homo Sapiens Chromosome X (HCX-L50)	30,977,493	50 bp	158,270,121	10
Homo Sapiens Chromosome X (HCX-L100)	15,488,736	100 bp	158,270,121	10
Homo Sapiens Chromosome X (HCX-L150)	10,326,314	150 bp	158,270,121	10
Homo Sapiens Chromosome X (HCX-L200)	7,744,119	200 bp	158,270,121	10
Homo Sapiens Chromosome 2 (HC2)	8,525,938	100 bp	245,653,507	10
Human Chromosome 14 (HC14)	36,504,800	101 bp	107,349,540	42
Staphylococcus Aureus (SA)	1,294,104	101 bp	2,903,081	45
Rhodobacter Sphaeroides (RS)	2,050,868	101 bp	4,603,060	45
speciesA_200i (A200i)	22,499,730	100 bp	N/A	40



	PPA-Assembler	Ray	SWAP-Assembler
16	214.93	8865.79	938.85
32	115.34	4878.28	470.08
48	90.23	3733.76	346.96
64	81.94	3013.38	291.46

Fig. 1. Running Time on HCX-C6 with Varying Number of Machines



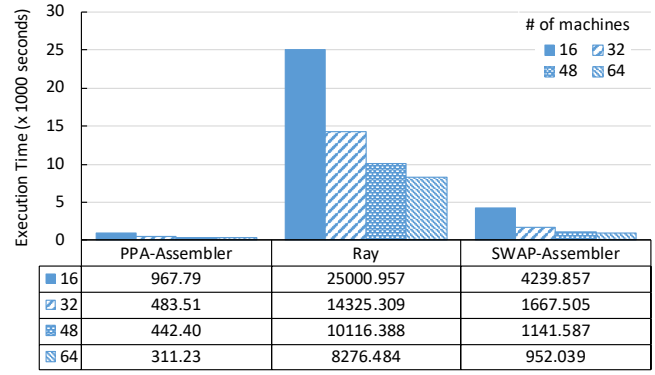
	PPA-Assembler	Ray	SWAP-Assembler
16	412.56	15694.10	2557.56
32	232.95	8434.25	1108.25
48	184.93	5982.17	813.51
64	172.47	5357.93	653.20

Fig. 2. Running Time on HCX-C10 with Varying Number of Machines

Specifically, we generate reads from NCBI's reference gene sequences Homo Sapiens Chromosome X (HCX)[3] and Homo Sapiens Chromosome 2 (HC2)[4], using the ART[5] software [11]. We set reader coverage as 10 and the average read length as 100 base pairs (bp) when generating the data. The generated datasets are HCX-C10 (or equivalently, HCX-L100) and HC2 in Table 1.

In order to test the assemblers' performance with different read depth and read length, we further generate reads from HCX reference gene sequence using read coverages 6 and 20, giving datasets HCX-C6 and HCX-C20 in Table 1; and we generate reads from HCX reference gene sequence using average read lengths 50 bp, 150 bp and 200 bp, giving datasets HCX-L50, HCX-L150 and HCX-L200 in Table 1.

3. http://www.ncbi.nlm.nih.gov/nuccore/NC_000023.11
4. https://www.ncbi.nlm.nih.gov/nuccore/NC_000002.12
5. http://www.niehs.nih.gov/research/resources/software/biostatistics/art/



	PPA-Assembler	Ray	SWAP-Assembler
16	967.79	25000.957	4239.857
32	483.51	14325.309	1667.505
48	442.40	10116.388	1141.587
64	311.23	8276.484	952.039

Fig. 3. Running Time on HCX-C20 with Varying Number of Machines

TABLE 2
Runtime of Single-Machine Assemblers with Varying Read Coverage

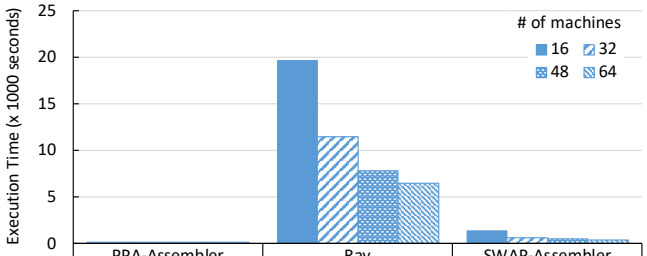
	Velvet	SPAdes	SOAPdenovo2
HCX-C6	1,271.8 s	1,326.8 s	631.9 s
HCX-C10	2,391.7 s	memory overflow	983.6 s
HCX-C20	15,482.7 s	memory overflow	memory overflow

To consider different species, we further incorporate the last 4 datasets shown in Table 1. The first 3 are downloaded from the GAGE project [21]: Human Chromosome 14 (HC14) [6], Staphylococcus Aureus (SA) [7], and Rhodobacter Sphaeroides (RS) [8]. The last dataset A200i is from the Assemblathon, which contains synthetic Illumina reads for species 'A' [9].

2.1 Running Time and Scalability

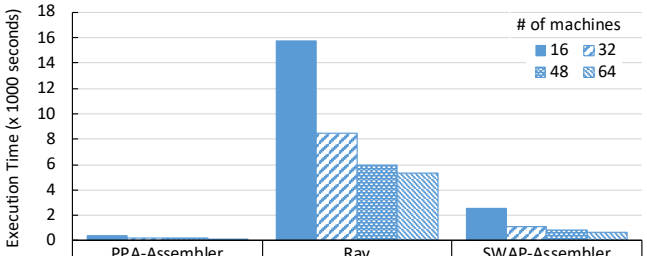
Efficiency with Different Read Coverage. Figures 1, 2 and 3 show running time of the distributed assemblers on the HCX datasets with read coverages 6, 10 and 20. For each assembler, we show the end-to-end execution time of assembly when each of our 15 machines runs 1, 2, 3 and 4 workers, respectively. We can see that for all read-coverage scenarios, PPA-Assembler is many times faster than SWAP-Assembler, while Ray is always around one order of magnitude slower than SWAP-Assembler. The performance of all systems improve with more workers and hence more parallelism. Abyss runs out of memory and is thus not reported.

6. http://gage.cbcb.umd.edu/data/Hg_chr14
7. http://gage.cbcb.umd.edu/data/Staphylococcus_aureus/
8. http://gage.cbcb.umd.edu/data/Rhodobacter_sphaeroides/
9. http://assemblathon.org/post/44431963352/assemblathon-1-data



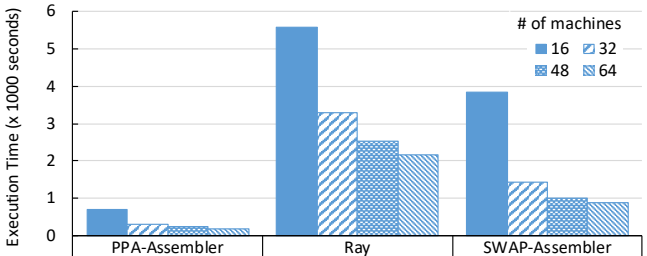
	PPA-Assembler	Ray	SWAP-Assembler
16	213.04	19602.14	1418.98
32	130.24	11510.32	625.56
48	102.93	7872.04	498.48
64	79.67	6532.83	455.91

Fig. 4. Running Time on HCX-L50 with Varying Number of Machines



	PPA-Assembler	Ray	SWAP-Assembler
16	412.56	15694.10	2557.56
32	232.95	8434.25	1108.25
48	184.93	5982.17	813.51
64	172.47	5357.93	653.20

Fig. 5. Running Time on HCX-L100 with Varying Number of Machines



	PPA-Assembler	Ray	SWAP-Assembler
16	718.82	5575.71	3829.62
32	313.71	3298.74	1423.57
48	246.57	2536.22	995.83
64	198.28	2167.22	876.10

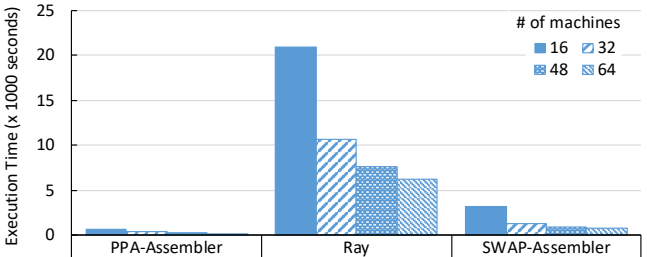
Fig. 6. Running Time on HCX-L150 with Varying Number of Machines

TABLE 3
Runtime of Single-Machine Assemblers with Varying Read Length

	Velvet	SPAdes	SOAPdenovo2
HCX-L50	1,386.5 s	memory overflow	425.1 s
HCX-L100	2,391.7 s	memory overflow	983.6 s
HCX-L150	2,077.4 s	memory overflow	482.8 s
HCX-L200	2,647.9 s	memory overflow	600.1 s

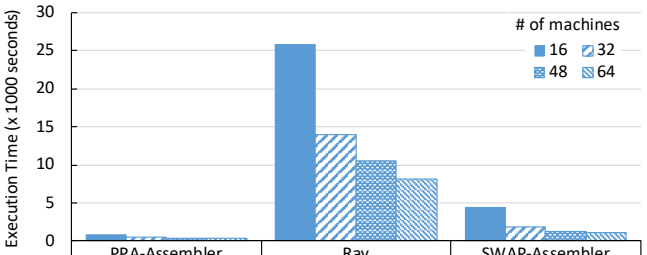
Table 2 shows the running time of the single-machine assemblers, where we can see that they are much slower than distributed assemblers and often run out of memory. This verifies the necessity and performance advantage of distributed assemblers.

Efficiency with Different Read Length. Figures 4, 5, 6 and 7 show the running time of distributed assemblers on the HCX datasets with average read lengths 50 bp, 100 bp, 150 bp and 200 bp. For each assembler, we show the end-to-end execution time of assembly when each of our 15 machines runs 1, 2, 3 and 4 workers, respectively. We can see that for all read-length scenarios,



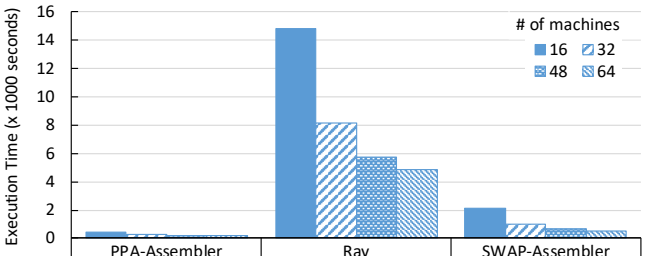
	PPA-Assembler	Ray	SWAP-Assembler
16	625.13	20998.89	3222.08
32	369.53	10707.20	1289.92
48	251.41	7686.07	967.56
64	215.04	6252.33	787.68

Fig. 7. Running Time on HCX-L200 with Varying Number of Machines



	PPA-Assembler	Ray	SWAP-Assembler
16	775.33	25758.16	4436.91
32	530.33	13940.41	1890.06
48	381.22	10523.63	1292.58
64	299.79	8147.76	1045.71

Fig. 8. Running Time on HC2 with Varying Number of Machines



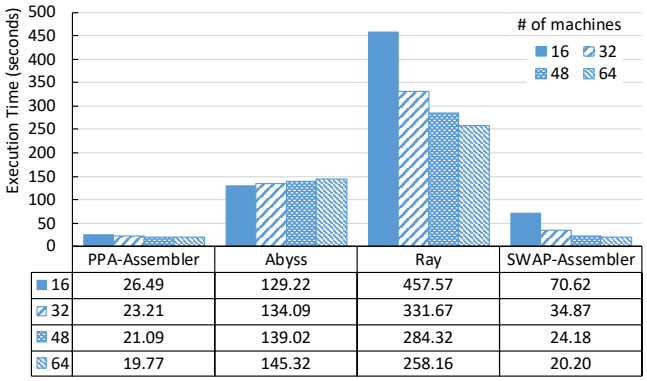
	PPA-Assembler	Ray	SWAP-Assembler
16	451.0	14786.249	2119.818
32	253.5	8132.725	967.672
48	221.1559	5758.513	659.305
64	159.3763	4841.82	551.333

Fig. 9. Running Time on HC14 with Varying Number of Machines

PPA-Assembler is many times faster than SWAP-Assembler, while Ray is always around one order of magnitude slower than SWAP-Assembler. The performance of all systems improve with more workers and hence more parallelism. Abyss runs out of memory and is thus not reported.

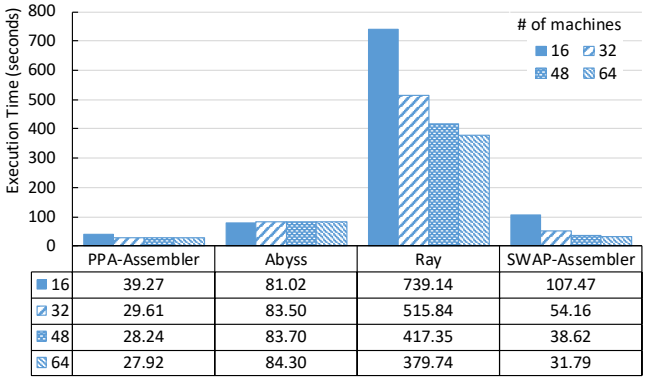
Table 3 shows the running time of the single-machine assemblers, where we can see that they are much slower than distributed assemblers and often run out of memory. This verifies the necessity and performance advantage of distributed assemblers.

Efficiency with Different Species. Figures 8, 9, 10, 11 and 12 show the running time of distributed assemblers on the datasets HC2, HC14, SA, RS and A200i. For each assembler, we show the end-to-end execution time of assembly when each of our 15 machines runs 1, 2, 3 and 4 workers, respectively. We can see that for all read-length scenarios, PPA-Assembler is many times faster than SWAP-Assembler and Abyss, while Ray is always around one order of magnitude slower than SWAP-Assembler.



	PPA-Assembler	Abyss	Ray	SWAP-Assembler
16	26.49	129.22	457.57	70.62
32	23.21	134.09	331.67	34.87
48	21.09	139.02	284.32	24.18
64	19.77	145.32	258.16	20.20

Fig. 10. Running Time on SA with Varying Number of Machines



	PPA-Assembler	Abyss	Ray	SWAP-Assembler
16	39.27	81.02	739.14	107.47
32	29.61	83.50	515.84	54.16
48	28.24	83.70	417.35	38.62
64	27.92	84.30	379.74	31.79

Fig. 11. Running Time on RS with Varying Number of Machines

Regarding the scalability with the number of workers, the performance of PPA-assembler, SWAP-Assembler and Ray keeps improving as the number of workers increases. In contrast, the performance of ABySS is insensitive to the number of workers. In fact, more workers may even lead to a longer assembly time. Abyss runs out of memory on HC2, HC14 and A200i and is thus not reported there.

Table 4 shows the running time of the single-machine assemblers, where we can see that they are much slower than distributed assemblers and often run out of memory. This verifies the necessity and performance advantage of distributed assemblers.

2.2 Sequencing Quality

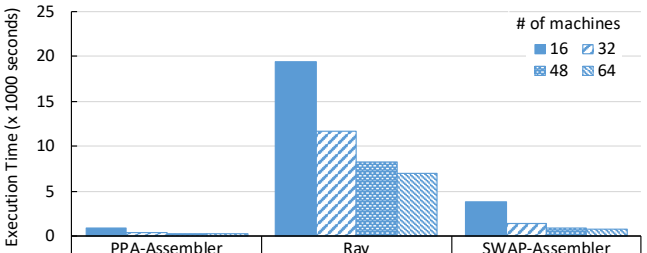
We now assess the sequencing quality of the assemblers. We remark that, for PPA-assembler, we are just evaluating the adopted workflow. We can easily configure PPA-assembler with other assembly strategies that lead to a higher sequencing quality. Even with the adopted workflow, PPA-assembler achieves comparable sequencing quality, which we present next.

We used the popular assessment tool, QUAST [10], which reports various quality metrics commonly used in genetic analysis. These metrics include: (1) N50, which is defined as the length for which the collection of all contigs of that length or longer covers at least half an assembly; (2) N7, which is similarly defined but with 75% instead of 50%; (3) # misassemblies, which is the number of positions in contigs (breakpoints) that correspond to misassemblies; (4) # misassembled contigs, which is the number of contigs that contain misassembly events.

We present the N50 results in Table 5, where entries with value "-" means that an assembler ran out of memory. We can see that PPA-assembler achieves N50 comparable to Ray, SWAP-Assembler and SOAPdenovo2, and is better on some datasets. Abyss and SPAdes achieve much higher N50, but they do not scale beyond small datasets. We also present the N75 results in Table 6, in which we obtain similar observations.

We remark that a higher N50 could be achieved due to more aggressive assembly strategies that may over-cleanse erroneous reads. For this purpose, we also study the numbers of misassemblies and misassembled contigs, which are shown in Tables 7 and 8, respectively. Note that a misassembled contig may contain multiple misassemblies. We can see that the much higher N50 of SPAdes is obtained in the sacrifice of accuracy, as its number of misassemblies is much higher than those of the other assemblers. In contrast, while Abyss can only scale to the two small datasts RA and RS, the accuracy looks good while achieving high N50.

Overall, PPA-assembler has a small number of misassemblies comparable to the other systems, and is a few times faster than even the second fastest assembler we tested, which verifies its good performance by utilizing the Pregel framework with cost-bounded Pregel algorithms. There is a great potential to extend PPA-assembler beyond its current basic assembly strategies to achieve a higher sequencing quality.



	PPA-Assembler	Ray	SWAP-Assembler
16	953.94	19379.09	3868.66
32	420.75	11745.56	1470.28
48	292.12	8296.28	929.72
64	231.58	7019.35	765.49

Fig. 12. Running Time on A200i with Varying Number of Machines

TABLE 4
Runtime of Single-Machine Assemblers with Different Species

	Velvet	SPAdes	SOAPdenovo2
HC2	18,542.3 s	3,057.7 s	memory overflow
HC14	1,896.0 s	1,377.0 s	809.6 s
Staphylococcus Aureus	34.8 s	68.4 s	53.7 s
Rhodobacter Sphaeroides	62.9 s	81.1 s	71.1 s
speciesA_200i	2,606.0 s	3239.1 s	843.4 s

TABLE 5
N50 of Different Datasets by Different Assemblers

	PPA	Abyss	Ray	SWAP	Velvet	SPAdes	SOAP
HCX-C6	1129	-	949	1131	1151	1343	1147
HCX-C10	1286	-	1051	1254	1688	-	1686
HCX-C20	1250	-	2489	1281	1597	-	-
HCX-L50	1265	-	1070	1438	963	-	966
HCX-L100	1286	-	1051	1254	1688	-	1686
HCX-L150	1026	-	2011	997	1834	-	1593
HCX-L200	1034	-	1562	971	1513	-	945
HC2	1221	-	1059	1116	1810	-	-
HC14	1244	-	1857	1236	1438	-	1379
SA	2615	2391	1313	1959	1525	5637	1088
RS	1743	2088	1389	1741	903	3935	1072
speciesA	1162	3431	2219	1159	1535	13738	1481

TABLE 6
N75 of Different Datasets by Different Assemblers

	PPA	Abyss	Ray	SWAP	Velvet	SPAdes	SOAP
HCX-C6	958	-	859	939	945	1032	946
HCX-C10	982	-	907	981	1168	-	1179
HCX-C20	988	-	1542	1005	1140	-	-
HCX-L50	1000	-	914	1026	866	-	867
HCX-L100	982	-	907	981	1168	-	1179
HCX-L150	890	-	1319	881	1229	-	1144
HCX-L200	889	-	1132	870	1099	-	854
HC2	951	-	909	937	1225	-	-
HC14	991	-	1276	983	1067	-	1042
SA	1615	1515	1019	1283	1123	3250	921
RS	1214	1397	1058	1198	841	2430	907
speciesA	952	2080	1479	947	1119	7847	1097

TABLE 7
Misassemblies of Different Datasets by Different Assemblers

	PPA	Abyss	Ray	SWAP	Velvet	SPAdes	SOAP
HCX-C6	0	-	0	1	19	9683	82
HCX-C10	1	-	3	10	1	-	16
HCX-C20	38	-	11	3634	0	-	-
HCX-L50	0	-	0	0	25	-	51
HCX-L100	1	-	3	10	1	-	16
HCX-L150	59	-	13	741	0	-	38
HCX-L200	41	-	19	372	0	-	2
HC2	2	-	3	15	0	-	-
HC14	16	-	22	1305	1	-	2
SA	0	0	0	24	0	6	0
RS	1	4	3	42	0	45	1
speciesA	no reference						

3 PRELMINARIES

Since PPA-assembler adopts the *de Bruijn graph* (DBG) based approach for sequencing [19], we build it on top of a Pregel-like distributed graph processing engine called Pregel+.

We next review the Pregel framework. For ease of presentation, we first define our graph notations. Given a graph $G = (V,E)$, we denote the number of vertices $|V|$ by n, and the number of edges $|E|$ by m. We also denote the *diameter* of G by δ.

If G is undirected, we denote the neighbors of a vertex v by $\Gamma(v)$ and the degree of v by $d(v) = |\Gamma(v)|$. If G is directed, we denote the in-neighbors (resp. out-neighbors) of v by $\Gamma_{in}(v)$ (resp. $\Gamma_{out}(v)$) and the in-degree (resp. out-degree) of v by $d_{in}(v) = |\Gamma_{in}(v)|$ (resp. $d_{out}(v) = |\Gamma_{out}(v)|$).

We denote the ID of v by $id(v)$, and use v and $id(v)$ interchangeably. Each vertex v also maintains a value denoted by $a(v)$.

Pregel [17]. In Pregel, vertices are distributed to different machines in a cluster, where each vertex v is associated with its adjacency list (e.g., $\Gamma(v)$) and its attribute $a(v)$. A program in Pregel implements a user-defined *compute*(.) function and proceeds in iterations (called *supersteps*). In each superstep, each active vertex v calls *compute*(*msgs*), where *msgs* is the set of incoming messages sent from other vertices in the previous superstep. In *v.compute*(*msgs*), v may process *msgs*, update $a(v)$, send new messages to other vertices, and vote to halt (i.e., deactivate itself). A halted vertex is reactivated if it receives a message in a subsequent superstep. The program terminates when all vertices are inactive and there is no pending message for the next superstep. Finally, the results (such as $a(v)$) are dumped to HDFS.

Pregel numbers the supersteps, so that a user may access the current superstep number in *compute*(.) to decide the proper behavior. Pregel also supports aggregator, a mechanism for global communication. Each vertex can provide a value to an aggregator in *compute*(.) in a superstep. The system aggregates those values and makes the aggregated result available to all vertices in the next superstep.

TABLE 8
Misassembled Contigs of Different Datasets by Different Assemblers

	PPA	Abyss	Ray	SWAP	Velvet	SPAdes	SOAP
HCX-C6	0	-	0	1	19	7794	81
HCX-C10	1	-	3	10	1	-	16
HCX-C20	38	-	11	3308	0	-	-
HCX-L50	0	-	0	0	25	-	51
HCX-L100	1	-	3	10	1	-	16
HCX-L150	58	-	13	683	0	-	38
HCX-L200	41	-	19	352	0	-	2
HC2	2	-	3	15	0	-	-
HC14	16	-	22	1206	1	-	2
SA	0	0	0	23	0	6	0
RS	1	4	3	41	0	43	1
speciesA	no reference						

Our Extensions to Pregel API. We find the following two extensions to the basic Pregel API very useful in implementing PPA-assembler. Firstly, for two consecutive jobs j and j', we allow j' to directly obtain input from the output of j in memory. In contrast, existing Pregel-like systems require j to first dump its output to HDFS, which is then loaded again by j'.

Let the vertex class of job j (resp. j') be V_j (resp. $V_{j'}$), then to enable the direct memory input, users need to define a user-defined function (UDF) *convert*(v) which indicates how to transform an object v of class V_j (processed by j) into zero or more input objects of class $V_{j'}$ (for job j'). After job j finishes, each machine generates a set of objects of type $V_{j'}$ by calling *convert*(.) on its assigned vertices of type V_j (which are then garbage-collected). Since Pregel+ distributes vertices to machines by hashing vertex ID, the generated objects of type $V_{j'}$ are then shuffled according to their vertex ID, before running job j'.

Secondly, the input data may not be in the format of one line per vertex. For example, each line may correspond to one edge, and hence the adjacency list of a vertex can be obtained from multiple lines. To create vertices from such input data, we support a mini-MapReduce procedure during graph loading.

Specifically, each line may generate zero or more key-value pairs (using UDF *map*(.)) where the key is vertex ID, and these key-value pairs are then shuffled according to vertex ID. After each machine receives its assigned key-value pairs, these pairs are sorted by key, so that all pairs with the same key form a group. Finally, each group with key $id(v)$ are processed (using UDF *reduce*(.)) to create the input vertex object v.

Practical Pregel Algorithm (PPA). Our prior work [28] defined a class of scalable Pregel algorithms called PPAs, and it designed PPAs for many fundamental graph problems. These PPAs can be used as building blocks to design PPAs for other sophisticated graph problems, such as DBG-based sequencing studied in this paper. Formally, a Pregel algorithm is called a *balanced practical Pregel algorithm* (BPPA) if it satisfies the following constraints:

1) *Linear space usage:* each vertex v uses $O(d(v))$ (or $O(d_{in}(v) + d_{out}(v))$ if G is directed) space of storage.
2) *Linear computation cost:* the time complexity of the $compute(.)$ function for each vertex v is $O(d(v))$ (or $O(d_{in}(v) + d_{out}(v))$ if G is directed).

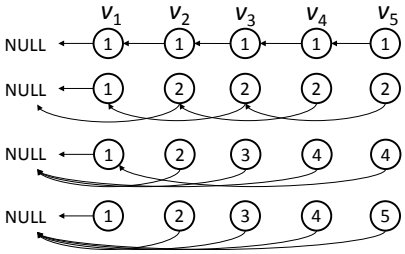


Fig. 13. Illustration of BPPA for List Ranking

3) *Linear communication cost:* at each superstep, the size of the messages sent/received by each vertex v is $O(d(v))$ (or $O(d_{in}(v)+d_{out}(v))$ if G is directed).
4) *At most logarithmic number of rounds:* the algorithm terminates after $O(\log n)$ supersteps.

Constraints 1-3 offers good load balancing and linear cost at each superstep, while Constraint 4 controls the total running time. Note that Constraint 4 includes those algorithms that run for a constant number of supersteps.

For some problems, the per-vertex requirements of BPPA can be too strict, and we can only achieve *overall linear space usage, computation and communication cost* (still in $O(\log n)$ rounds). We call a Pregel algorithm that satisfies these constraints simply as a *practical Pregel algorithm* (PPA). Workload skewness of PPA can be solved using the request-respond API of Pregel+ [27].

We now review two PPAs proposed in [28], both will be used by PPA-assembler for finding contigs in Section 4.2.

BPPA for List Ranking. Consider a linked list $\mathscr{L}$ with n vertices, where each vertex v keeps a value $val(v)$ and its predecessor $pred(v)$. The vertex v at the head of $\mathscr{L}$ has $pred(v) = null$.

For each vertex v in $\mathscr{L}$, let us define $sum(v)$ to be the sum of the values of all the vertices from v following the predecessor link to the head. The *list ranking* problem computes $sum(v)$ for every vertex v in $\mathscr{L}$, where the vertices are stored on HDFS in arbitrary order.

The BPPA for list ranking works as follows. Each vertex v initializes $sum(v) \leftarrow val(v)$. Then in each round, each vertex v does the following in *compute*(.): if $pred(v) \neq null$, v sets $sum(v) \leftarrow sum(v) + sum(pred(v))$ and $pred(v) \leftarrow pred(pred(v))$; otherwise, v votes to halt. Note that to perform these updates, v needs to first request its predecessor $w = pred(v)$ for $sum(w)$ and $pred(w)$, which takes another superstep. This process repeats until $pred(v) = null$ for every vertex v, at which point all vertices vote to halt and we have $sum(v)$ as desired.

Figure 13 illustrates how the algorithm works. Initially, objects v_1–v_5 form a linked list with $sum(v_i) = val(v_i) = 1$ and $pred(v_i) = v_{i-1}$. Let us now focus on v_5. In Round 1, we have $pred(v_5) = v_4$ and so we set $sum(v_5) \leftarrow sum(v_5) + sum(v_4) = 1+1 = 2$ and $pred(v_5) \leftarrow pred(v_4) = v_3$. One can verify the states of the other vertices similarly. In Round 2, we have $pred(v_5) = v_3$ and so we set $sum(v_5) \leftarrow sum(v_5) + sum(v_3) = 2+2 = 4$ and $pred(v_5) \leftarrow pred(v_3) = v_1$. In Round 3, we have $pred(v_5) = v_1$ and so we set $sum(v_5) \leftarrow sum(v_5) + sum(v_1) = 4+1 = 5$ and $pred(v_5) \leftarrow pred(v_1) = null$. The number of vertices whose values get summed is doubled after each iteration, and thus the algorithm terminates in $\log n$ rounds.

Simplified S-V Algorithm. The S-V algorithm was proposed in [28] for computing the connected components (CCs) of a big undirected graph G in $O(\log n)$ number of supersteps, by adapting Shiloach-Vishkin's PRAM algorithm [23] to run in Pregel.

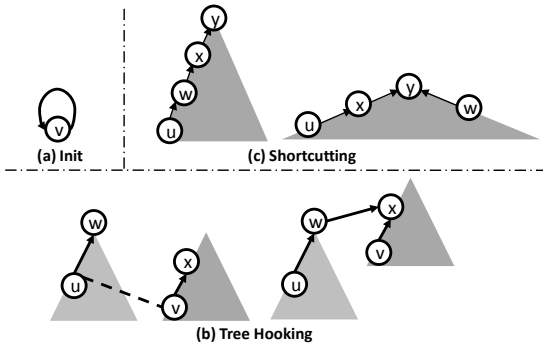


Fig. 14. Illustration of the Simplified S-V Algorithm

In the S-V algorithm, each round of computation requires three operations: tree hooking, star hooking, and shortcutting. However, we find that star hooking is actually an artifact required by the original Shiloach-Vishkin's algorithm for correct termination in the PRAM setting. Here, we propose a simplified version of the S-V algorithm that does not require star hooking, which is more efficient as the expensive checking of whether a vertex is in a star (i.e., a tree with height 1) required by the original S-V algorithm is eliminated.

Throughout this algorithm, vertices are organized by a forest such that all vertices in a tree belong to the same CC. Each vertex v maintains a link $D[v]$ to its parent in the forest. We relax the tree definition a bit here to allow the tree root w to have a self-loop (i.e., $D[w] = w$).

At the beginning, each vertex v initializes $D[v] \leftarrow v$, forming a self loop as shown in Figure 14(a). Then, the algorithm proceeds in rounds, and in each round, the parent links are updated in two steps: (1) *tree hooking* (see Figure 14(b)): for each edge (u,v), if u's parent $w = D[u]$ is a tree root and $D[v] < w$, we hook w as a child of v's parent $x = D[v]$ (i.e., we merge the tree rooted at w into v's tree); (2) *shortcutting* (see Figure 14(c)): for each vertex v, we move it closer to the tree root by linking v to the parent of v's parent, i.e., $D[D[v]]$. Note that Step 2 has no impact on $D[v]$ if v is a root or a child of a root.

The algorithm repeats these two steps until no vertex v has $D[v]$ updated in a round (checked by using aggregator), by which time every vertex is in a star, and each star corresponds to a CC. Since $D[v]$ monotonically decreases during the computation, at the end $D[v]$ equals the smallest vertex in v's CC (which is also the root of v's star). In other words, all vertices with the same value of $D[v]$ constitute a CC.

The algorithm is a PPA since (1) each round can be formulated in Pregel as a constant number of supersteps, and (2) shortcutting allows the algorithm to run in $O(\log n)$ rounds. The proof of Conclusion (2) is non-trivial and is only recently solved by strong theoretical computer scientists Sixue Liu and Robert E. Tarjan in their work of [13], who noticed a flaw in proving the logarithmic-round bound of our prior SV algorithm proposed in [28]. We refer interested readers to Section 5.3 of [13] for the proof.

4 The Design of PPA-Assembler

This section presents the workflow of PPA-Assembler and the Pregel algorithms of its operations. We assume that readers are already familiar with the concepts in DGB-based de novo genome assembly such as reads, contigs, k-mers, reverse complement, tips

Fig. 15. DNA Strands

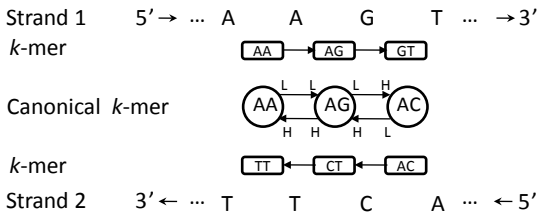


Fig. 16. Canonical k-mers & Edge Polarity

and bubbles. If they are new to you, please first refer to Section III of our arXiv preprint [26] for a detailed tutorial.

4.1 Directionality and Data Format

Directionality. Directionality arises since reads may be obtained from both strands of the DNA molecule. As Figure 15 shows, each strand has an end-to-end chemical orientation, and reads are always obtained in the 5'-to-3' direction. Specifically, strand 1 (resp. strand 2) is read from left to right (resp. from right to left). Given a nucleotide x, we denote its complement by $\overline{x}$. The *reverse complement* of a DNA sequence $s = x_1 x_2 \ldots x_\ell$ is denoted by $rc(s) = \overline{x_\ell}\ \overline{x_{\ell-1}} \ldots \overline{x_1}$ (or simply $\overline{x_\ell x_{\ell-1} \ldots x_1}$). For example, the reverse complement of strand 1 in Figure 15 is "GACTTGCAAT", which is exactly strand 2 reading in the 5'-to-3' direction.

Now consider the highlighted read "AAGT" in Figure 15 which is re-plotted on strand 1 in Figure 16. If we read the same DNA segment on strand 2 in the 5'-to-3' direction, we obtain another read "ACTT", which is exactly the reverse complement of "AAGT". Figure 16 also shows the k-mer vertices and $(k+1)$-mer edges generated by these two reads ($k = 2$).

We would like a k-mer and its reverse complement to correspond to a unique vertex in the DBG, so that reads from different strands can be stitched to create longer contigs as long as the reads share overlapping DNA segments. To achieve this goal, we define the *canonical k-mer* of a k-mer s as the lexicographically smaller sequence between s and $rc(s)$, and use the canonical k-mer as a vertex in the DBG. For example, the rightmost k-mers "GT" and "AC" in Figure 16 both refer to the rightmost DBG vertex "AC" of the chain in the middle of Figure 16.

Accordingly, now each DBG edge (u,v) needs to have a *polarity* to indicate the direction of a $(k+1)$-mer that generates this edge, i.e., u-to-v or v-to-u. Polarity is used to indicate the stitching directions when constructing contigs. We use an example to explain how edge polarity is determined. Consider the last $(k+1)$-mer of read "AAGT" from strand 1 in Figure 16 ($k = 2$), i.e., "AGT", which creates an edge "AG"→"GT". Edge source "AG" is already canonical and thus we give it a label L, while edge target "GT" needs to be converted to its reverse complement "AC" to be a DBG vertex, in which case we give it a label H. The edge direction is simply a concatenation of the source and target labels, i.e., $\langle L:H \rangle$. We say that labels H and L are complementary, and denote $\overline{H} = L$ and $\overline{L} = H$. It is not difficult to see the following property (e.g., from Figure 16).

Property 1. Edge (u,v) with polarity $\langle X:Y \rangle$ is equivalent to edge (v,u) with polarity $\langle \overline{Y}:\overline{X} \rangle$.

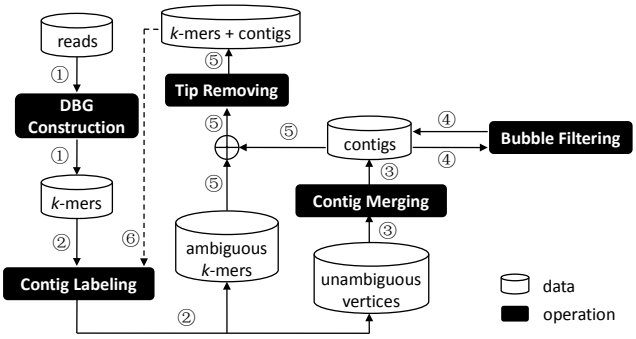


Fig. 17. Workflow of PPA-Assembler Operations

This property allows us to stitch k-mers generated from different strands. For example, consider $(k+1)$-mers "AAG" from strand 1 and "ACT" from strand 2, which generates two edges "AA"$\xrightarrow{\langle L:L \rangle}$"AG" and "AC"$\xrightarrow{\langle L:H \rangle}$"AG". Although both edges are incident to "AG", the labels at the side of "AG" do not match. Since the latter edge is equivalent to "AG"$\xrightarrow{\langle L:H \rangle}$"AC", we can stitch the edges to obtain "AA"$\xrightarrow{\langle L:L \rangle}$"AG"$\xrightarrow{\langle L:H \rangle}$"AC" where both edges agree on label L for "AG" and are in the same direction.

Memory-Efficient Scheme of Storing Vertices and Edges. We design compact data structures for vertices and edges in our vertex-centric programs to be **memory-efficient**, since genome assembly has a very high memory demand [12]. To keep the presentation succinct, we only highlight some key designs, and the complete description on the data structure of vertices and edges can be found in Section IV-A of our arXiv preprint [26].

Each vertex in a Pregel program has a unique ID for message passing, and we use integer to specify vertex ID for efficiency reasons. There are two kinds of vertices in PPA-Assembler, (1) ***k*-mer** and (2) **contig**. We encode the sequence of a k-mer directly into its integer ID, where each nucleotide is represented by two bits: A (00), T (11), G (10), C (01). A 64-bit ID can keep up to 31 nucleotides, with the 2 most significant bits reserved. In contrast, a contig can be an arbitrarily long sequence which has to be kept as a vertex attribute. For vertex ID, the i-th worker machine assigns its j-th generated contig a 64-bit ID that equals the 32-bit integer representation of i (usually only using the least significant few bits) concatenated with the 32-bit integer representation of j. The 2 most significant bits are used to indicate whether a vertex is a k-mers, or a contig, or a dummy vertex *NULL* used to indicate that a k-mer or a contig has no neighbor along one direction.

We also encode the neighbors of a k-mer vertex with a 32-bit bitmap. For example, the 4-mer "CCGT" can have at most 4 in-neighbors whose suffix matches its prefix "CCG", i.e., "ACCG", "TCCG", "GCCG" and "CCCG". Similarly, there are at most 4 out-neighbors. Now taking the 4 possible edge polarity $\langle L:L \rangle$, $\langle L:H \rangle$, $\langle H:L \rangle$ and $\langle H:H \rangle$ into account, we obtain $4 \times (4+4) = 32$ possible combinations, which we represent using the 32-bit bitmap. A bit is 1 only if the corresponding neighbor exists.

A k-mer vertex tracks its contig neighbors differently from the k-mer neighbors. Let a contig w be connected to two k-mers u and v on its two ends, then u keeps v as its neighbor instead but with the edge labeled by w's vertex ID, so that u can request w's sequence using the ID. Please refer to Section IV-A of our arXiv preprint [26] for details.

Vertex Types. First consider a k-mer vertex v, and it can be of one of the following three types: (1) $\langle 1 \rangle$: such a vertex only has

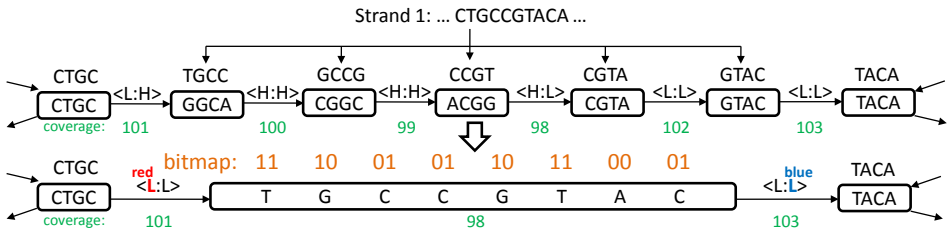


Fig. 18. Contig Format

one neighbor, and is thus a dead-end; (2) $\langle$1-1$\rangle$: such a vertex has two neighbors, and when both edges agree on the polarity label for v (either L or H) which can be enforced using Property 1, one neighbor is an in-neighbor and the other is an out-neighbor; such a vertex is unambiguous; (3) $\langle m\text{-}n\rangle$: such a vertex has at least two neighbors, but it does not satisfy the requirement of $\langle$1-1$\rangle$; such a vertex is ambiguous. Note that v must have a neighbor, since a k-mer vertex is contributed by the prefix or suffix of a $(k+1)$-mer.

Since a contig is generated by merging unambiguous k-mers, it can only be of type $\langle 1\rangle$ or type $\langle$1-1$\rangle$. Here, we say that a contig vertex is of type $\langle 1\rangle$ iff at least one of its two neighbors is *NULL*, i.e., the contig corresponds to a dangling path in DBG and is thus a tip candidate (depending on the contig length).

4.2 The Workflow of Operations and Their Algorithms

PPA-assembler provides a library of operations for flexible genome assembly in a distributed environment deployed with Hadoop. Each operation is implemented as a PPA (described in Section 3) and is thus scalable; it can either load data from HDFS, or directly obtain input from another operation's output in memory. Users may combine the provided operations to implement various sequencing strategies, and they may even integrate new operations or re-define existing operations (e.g., by changing the criteria for judging tips and bubbles) using the convenient vertex-centric API of Pregel+.

Overview. Figure 17 shows the data flow diagram of PPA-assembler, which includes five operations: ① *DBG construction*, which constructs a DBG from the DNA reads, and outputs the k-mer vertices of the DBG along with their adjacency lists; ② *contig labeling*, which divides the vertices into two sets (ambiguous ones and unambiguous ones) and labels unambiguous vertices by the contigs that they belong to; ③ *contig merging*, which merges unambiguous vertices into contigs according to the labels; ④ *bubble filtering*, which filters any low-coverage contig that shares both ends with another high-coverage contig that has a similar sequence; ⑤ *tip removing*, which takes the ambiguous k-mers and the contigs (after bubble filtering), and removes tips.

In fact, the output of tip removing can be fed to the "contig labeling" operation again to grow longer contigs (see arrow ⑥ in Figure 17), since the previous error correction operations may have converted some ambiguous k-mer vertices into unambiguous ones, and the operations ②–⑤ may loop as many times as needed (though we typically just loop for one more round). At the first round, the inputs to operations "contig labeling" and "contig merging" must be k-mers, but starting from the second round, the inputs may contain a mix of k-mers and contigs. For ease of discussion, we focus on the first round when discussing operations "contig labeling" and "contig merging".

① DBG Construction. This operation loads DNA reads from HDFS, and creates a DBG from them through two mini-MapReduce phases: (i) the first phase extracts $(k+1)$-mers from reads, and (ii) the second phase constructs k-mer vertices and their adjacency lists from the extracted $(k+1)$-mers, to form the DBG.

We first describe phase (i). In real DNA data, the sequence of a read may contain element "N" besides "A", "T", "G", "C", and such an element indicates that the nucleotide cannot be determined due to noise in measurement. For this purpose, in *map(.)*, a read is first split into sequences by elements "N", and each sequence is parsed to obtain the $(k+1)$-mers using a sliding window of $(k+1)$ elements. The sequence of a $(k+1)$-mer is directly encoded in its 64-bit integer ID, which functions as the key for shuffling. In each worker machine, if a $(k+1)$-mer is obtained for the first time, the worker creates an $(ID, count)$ pair for it where $counter = 1$; otherwise, the counter of the $(k+1)$-mer is incremented by 1. After shuffling, for each $(k+1)$-mer, all its counts (from all workers) are input to *reduce(.)*, which then sums these counts to obtain the total count of the $(k+1)$-mer; *reduce(.)* only outputs the $(k+1)$-mer as an $(ID, count)$ pair, if the coverage $count > \theta$ where θ is a user-defined threshold. We filter a low-coverage $(k+1)$-mer since it is very likely to be contributed by erroneous readers.

In Phase (ii), each remaining $(k+1)$-mer is input to *map(.)*, which extracts two k-mers that correspond to its prefix and suffix. In each worker, if an extracted k-mer is obtained for the first time, the worker creates a k-mer vertex for it. A directed edge from the prefix k-mer vertex to the suffix k-mer vertex is also added into the adjacency lists of both k-mer vertices (recall that an adjacency list is represented by a 32-bit bitmap), and edge count (which equals the $(k+1)$-mer's count) is also recorded or incremented, using a variable-length integer. The k-mer vertices with partially constructed adjacency lists are then shuffled by the 64-bit integer ID. After shuffling, for each k-mer, its partial adjacency lists (from all workers) are input to *reduce(.)*, which combines them to obtain the complete adjacency list (still represented in 32 bits), and which sums the edge counts for each edge to obtain the edge's coverage (represented compactly using a variable-length integer).

② Contig Labeling. Let us call a path that only contains vertices of types $\langle 1\rangle$ and $\langle$1-1$\rangle$ as an unambiguous path. The "contig labeling" operation marks all vertices on each maximal unambiguous path with a unique label, so that they can be grouped to create a contig later. The operation is executed right after "① DBG construction", and the input vertices are all k-mers. It can also be executed after "⑤ tip removing" to find longer contigs, in which case some input vertices could already be contigs.

A vertex is at one end of a maximal unambiguous paths, if its type is $\langle 1\rangle$, or if its type is $\langle$1-1$\rangle$ and at least one neighbor is of type $\langle m\text{-}n\rangle$ (i.e., ambiguous). The contig labeling operation first recognizes contig-ends in two supersteps: (1) in superstep 1, every vertex of type $\langle m\text{-}n\rangle$ broadcasts its ID to all its neighbors, and then votes to halt; it will never be reactivated again as the remaining computation only involves unambiguous vertices; (2) in

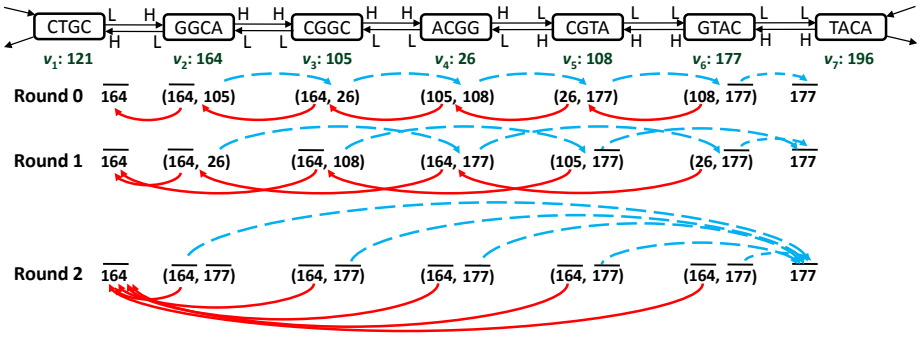


Fig. 19. Bidirectional List Ranking

superstep 2, a vertex recognizes itself as a contig-end if it is of type $\langle 1 \rangle$, or if it is of type $\langle 1\text{-}1 \rangle$ and receives the ID of any ambiguous vertex sent from superstep 1.

There are two methods to find all maximal unambiguous paths (i.e., contigs) in $O(\log n)$ supersteps, both of which require contig-end vertices to remove all their edges that connect to ambiguous vertices, so that the DBG graph becomes a set of isolated unambiguous paths, each corresponding to a contig. The first method is to run the simplified S-V algorithm described in Section 3, so that every vertex v is labeled with the smallest vertex ID in its connected component (i.e., isolated unambiguous path containing v). The second method is to use the idea of list ranking described in Section 3 to find all unambiguous paths in $O(\log \ell_{max})$ time, where ℓ_{max} is the length of the longest unambiguous path.

We now describe the second algorithm in more detail. We illustrate this algorithm using the example of Figure 18, which is re-plotted in Figure 19. Each edge is plotted along with its equivalent edge in the other direction as determined by Property 1, and each vertex is denoted by its integer ID (e.g., "GGCA" is encoded with bitmap 10100100, which is 164).

As mentioned three paragraphs before, in superstep 2, a vertex v that recognizes itself as a contig-end needs to remove edges that connect with any ambiguous vertex. Instead of deleting such an edge from v's adjacency list, we replace it with a self-loop edge, but we flip the second most significant bit of the edge's target ID (i.e., $id(v)$) to indicate that v is a contig-end (recall that the two most significant bits are reserved). The flipped ID is denoted by $\overline{id(v)}$. For example, in Figure 19, vertex v_2 with ID 164 has two neighbors v_1 and v_3, and it replaces the edge that connects with the ambiguous neighbor v_1 (who sent its ID to v_2 in superstep 1) by a self-loop edge to v_2 itself, leading to a pair of neighbor IDs $(\overline{164}, 105)$.

In our list ranking approach, each unambiguous vertex maintains a pair of IDs, which is initialized as the pair of neighbor IDs set by superstep 2, as illustrated by round 0 in Figure 19. Note that a vertex v of type $\langle 1 \rangle$ also has a pair of IDs, since its *NULL* neighbor is replaced with the self-loop edge (note that v is a contig-end). We then perform list ranking in both sequencing directions of a contig, and we call the process as *bidirectional list ranking*. We pass messages in both directions rather than from one end of a contig to the other end, since the two ends are symmetrically recognized in superstep 2, and edge direction alone is not sufficient to determine the sequencing direction as explained by Property 1.

In the ID pair maintained by a vertex v, each ID corresponds to v's predecessor in one sequencing direction, which is updated as the predecessor's predecessor after each round until it becomes the flipped ID of a contig-end. We illustrate this process by considering vertex v_3 with ID 105 in Figure 19. In round 0, v_3 sends its ID to its two predecessors 164 (v_2) and 26 (v_4) in one superstep, to request for their predecessors. In the next superstep, v_4 receives v_3's ID 105, checks its own ID pair $(105, 108)$, and responds the predecessor $v_5 = 108$ back to v_3 since the other value 105 in the ID pair is the received requester's ID.

Similarly, v_3 will also receive $\overline{164}$ from v_2. It then replaces its current ID pair with the two received predecessor IDs $(\overline{164}, 108)$.

In round 1, v_3 sends requests to $v_5 = 108$ only since it has already reached the contig-end $\overline{164}$ in the other direction. It receives v_5's predecessor $\overline{177}$, and updates its ID pair as $(\overline{164}, \overline{177})$. Since it reaches both contig-ends, it votes to halt and will not participate in any future computation. In fact, all vertices reach both contig ends before round 2 and vote to halt, and thus the computation stops in 2 rounds. It is not difficult to see that the number of hops between a vertex and its precedessors gets doubled by each round, and thus the algorithm stops in $O(\log \ell_{max})$ supersteps and is thus a BPPA. When the computation terminates, the ID pair of each vertex contains the flipped IDs of its two contig-ends. Obviously, each ID pair uniquely defines a contig, and we use the smaller one of the pair of contig-end vertex IDs as the contig-label.

Bidirectional list ranking alone is not sufficient if the DBG contains a cycle of vertices of type $\langle 1\text{-}1 \rangle$ (a special contig if large enough), since these vertices will never reach an end; in contrast, for a normal case each round will have some vertices vote to halt due to reaching both contig-ends. Based on these observations, if the number of active vertices is larger than 0 but does not decrease after a round, our algorithm turns to run our simplified S-V algorithm on the remaining active vertices, so that each vertex in a cycle obtains the smallest ID in the cycle. Bidirectional list ranking is preferred since each round only takes 2 supersteps, much smaller than that required by a round of the S-V algorithm. On the other hand, running the S-V algorithm over vertices in cycles at last is fast, since there are very few active vertices remaining.

To summarize, if only the simplified S-V algorithm is adopted, then each vertex obtains its contig-label as the smallest vertex ID in its contig; if bidirectional list ranking is adopted, each vertex in a non-cycle contig obtains its contig-label as the smaller contig-end vertex's ID, while each vertex in a cycled contig obtains its contig-label as the smallest vertex ID in the cycle; it is also a must to ensure the correctness of the algorithm.

③ **Contig Merging.** This operation takes the labeled unambiguous vertices as the input, and uses a mini-MapReduce procedure to group the vertices by their labels. All vertices with the same contig-label are input to *reduce*(.), which then merges the sequences of these vertices to obtain the contig.

We now describe the merging process in *reduce*(.). Firstly, a hash table is constructed over all the vertices in the contig-group, so that we can lookup a vertex object (storing information like its sequence and neighbors) using its 64-bit integer ID. We also identify a contig-end vertex, which contains a neighbor not in the group (either *NULL* or of type $\langle m\text{-}n\rangle$), to start the stitching with. If such a vertex cannot be found, the contig is cycled and we start stitching from an arbitrary vertex.

We then order all the vertices from the starting vertex (and meanwhile, set the edge directions properly), so that they can be stitched in order. Let us denote the starting vertex by v_1, and denote the subsequent vertices after ordering by $v_2, v_3, \ldots, v_k$. Initially, we find a neighbor of v_1 that is not its self-loop, which is found as v_2. We let v_1's out-neighbor be v_2, and let the other neighbor of v_1 be its in-neighbor. Edge directions and polarities are properly adjusted using Property 1 if they are originally inconsistent. We then obtain v_2 from the hash table for processing, using its ID stored in v_1's adjacency list. Generally, for each vertex v_i $(i > 1)$, we let v_{i-1} be its in-neighbor, and let the other neighbor (which is found as v_{i+1}) be the out-neighbor; then v_{i+1} can be obtained from the hash table (using its ID in v_i's adjacency list) to continue the ordering process. The ordering finishes when all k vertices have been processed.

If v_k is of type $\langle 1\rangle$, we exit *reduce*(.) if the aggregated contig length is not above the user-specified tip-length threshold (since the contig is a tip). In all other cases, we stitch the vertices in the order of $v_1, v_2, \ldots, v_k$ to construct the contig's sequence. We also set the contig's coverage as the minimum edge coverage seen during the concatenation, and set the contig's two neighbors with v_1's in-neighbor and v_k's out-neighbor.

④ **Bubble Filtering.** The contigs previously constructed may then enter the "bubble filtering" operation for further filtering though a mini-MapReduce procedure. In *map*(.), each contig with neighbors nb_1 and nb_2 $(nb_1 < nb_2)$, both of type $\langle m\text{-}n\rangle$, associates itself with a key (nb_1, nb_2) for shuffling. As a result, all contigs that share two neighboring ambiguous vertices (nb_1, nb_2) are input to *reduce*(.), and let us denote them by $c_1, c_2, \ldots, c_k$. We then process each contig c_i as follows: if c_i is not already pruned, we check whether any contig c_j $(j > i)$ can prune c_i. Specifically, we first compute the edit distance between c_i's sequence and c_j's sequence or its reverse complement (depending on whether c_i and c_j's edge directions are consistent, i.e., nb_1-to-nb_2 or nb_2-to-nb_1). If the distance is smaller than a user-defined threshold, we mark c_i (resp. c_j) as pruned if its coverage is smaller than c_j (resp. c_i).

⑤ **Tip Removing.** This operation takes both ambiguous k-mers and the merged contigs as input. We first need to update the adjacency lists of the ambiguous k-mers, to link them to the newly merged contigs. In fact, since some contigs may have been removed due to bubble filtering, some ambiguous k-mers may have changed their types from $\langle m\text{-}n\rangle$ to $\langle 1\text{-}1\rangle$ or $\langle 1\rangle$. Recall from Section 4.1 that a k-mer vertex stores its contig neigbhor by maintaining (1) the contig vertex's ID (e.g., for requesting its sequence), (2) the vertex that the contig connects to on the other end, and (3) other contig information like its length. We set the adjacency lists of the k-mer vertices in two supersteps: (i) in superstep 1, each contig vertex sends its information mentioned above to both neighbors (if not *NULL*); then (ii) in superstep 2, each k-mer vertex collects these information into its adjacency list.

Since only path length is concerned during tip removing, we only need to check the k-mer vertices since each k-mer vertex u maintains the sequence length of each contig neighbor c that connects to v. However, when deleting the edge (u, v) (due to being part of a tip), a message should be sent to the contig vertex c (using c's ID) to tell it to delete itself, which we take for granted and will not emphasize in the subsequent algorithm description.

Note that the removal of tips may cause some vertices of type $\langle m\text{-}n\rangle$ to change their type to $\langle 1\rangle$, hence generating new tips. We thus run vertex-centric tip removing for multiple phases, until no new $\langle 1\rangle$-typed vertex is generated at the end of a phase.

In a phase, we start message passing from vertices of type $\langle 1\rangle$, where a message records (1) the sender's ID, (2) cumulative sequence length, and (3) a type *REQUEST*. A $\langle 1\rangle$-typed vertex u initializes the cumulative sequence length as k (i.e., u's k-mer sequence length). When a vertex u of type $\langle 1\text{-}1\rangle$ receives a *REQUEST* message, it relays the message to the other neighbor v (which is not the sender) by adding the cumulative sequence length by 1 if u is a k-mer vertex, and by $1 + len(c) - (k-1)$ if u is a contig c, where 1 is contributed by u, $len(c)$ is the contig length, and we subtract $(k-1)$ from the length to avoid double-counting the overlapping nucleotides already counted in the cumulative sequence.

The *REQUEST* message ends at an $\langle m\text{-}n\rangle$-typed or $\langle 1\rangle$-typed vertex v, which checks whether the cumulative sequence length is not larger than the tip-length threshold. If so, v sends a message of type *DELETE* to the sender to delete the vertices on the dangling path. The *DELETE* message is relayed by $\langle 1\text{-}1\rangle$-typed vertices back till reaching the $\langle 1\rangle$-typed vertex that initiates the *REQUEST* message, and vertex and contig deletions are triggered along the backward message propagation.

A special case is when a tip has two $\langle 1\rangle$-typed ends. Since both vertices at the ends initiate a *REQUEST* message sent towards each other, when the two *DELETE* messages are sent back, they meet in the middle of the tip (rather than reach the other $\langle 1\rangle$-typed end).

An $\langle m\text{-}n\rangle$-typed vertex v also deletes its edge to the neighbor that it sends a *DELETE* message, and if its type becomes $\langle 1\rangle$, it keeps itself activated to initiate the *REQUEST* message in the next phase.

5 Additional Experiments

Recall from Section 3 and Section 4.2 that there are two approaches for contig labeling (see Step ② in Figure 17), bidirectional list ranking and simplified S-V. As we mentioned, while both algorithms are PPAs that run for $O(\log n)$ rounds (and hence supersteps), each round of S-V require a larger number of supersteps than a round in list ranking, and thus list ranking (LR) is expected to be much faster.

Recall that we run PPA-assembler with the simple workflow of ①②③④⑤⑥②③ in Figure 17, and "② contig labeling" is performed twice: once for labeling unambiguous k-mers, and once for labeling contigs (to grow longer ones).

We have seen in Section 2 that PPA-assembler is from a few times to tens of times faster than existing state-of-the-art assemblers. There, we used the bidirectional list ranking algorithm for contig labeling (see Step ③ in Figure 17). This section justifies our choice by comparing it with if we adopt the (simplified) S-V algorithm for contig labeling.

Table 9 and Table 10 show the comparison of LR and S-V for labeling k-mers and labeling contigs, respectively, on all the datasets. There, we report (1) the number of supersteps, (2) the number of messages, and (3) the running time.

TABLE 9
Bidirectional LR v.s. S-V for Labeling Unambiguous k-mers

Datasets	# of Supersteps		# of Messages		Runtime (s)	
	LR	SV	LR	SV	LR	SV
HCX-C6	50	86	573,558,630	2,524,607,382	35.97	131.71
HCX-C10 / L100	43	86	1,788,475,951	6,380,722,570	104.06	346.32
HCX-C20	28	93	4,065,836,071	10,384,170,426	271.43	635.03
HCX-L50	50	79	587,710,981	3,064,451,778	39.92	165.90
HCX-L150	36	93	3,189,223,427	9,203,242,886	216.92	547.78
HCX-L200	43	93	2,837,913,137	8,477,515,372	187.86	514.11
HC2	50	79	2,920,014,695	9,435,316,157	200.39	591.73
HC14	50	93	2,338,866,944	6,147,451,231	151.07	360.25
RS	22	86	136,360,533	323,144,002	7.15	15.12
SA	43	93	100,617,701	220,871,422	5.30	10.17
A200i	43	114	3,332,818,876	11,309,323,655	223.07	670.95

TABLE 10
Bidirectional LR v.s. S-V for Labeling Contigs

Datasets	# of Supersteps		# of Messages		Runtime (s)	
	LR	SV	LR	SV	LR	SV
HCX-C6	22	44	860,786	3,863,872	0.35	0.48
HCX-C10 / L100	22	37	1,810,183	6,384,157	0.46	0.61
HCX-C20	15	37	3,357,356	4,916,223	0.55	0.63
HCX-L50	22	51	971,506	5,205,241	0.35	0.59
HCX-L150	22	37	2,617,918	5,134,871	0.54	0.63
HCX-L200	22	37	2,385,926	5,234,668	0.49	0.58
HC2	22	44	2,372,566	8,958,344	0.53	0.85
HC14	29	44	1,650,716	4,648,384	0.48	0.53
RS	8	51	82,072	436,652	0.18	0.34
SA	15	37	17,286	58,827	0.20	0.24
A200i	22	37	1,563,546	8,768,339	0.41	0.68

We can see that LR runs for much fewer supersteps, sends much fewer messages, and is much faster than S-V. The message number and running time in Table 10 are three orders of magnitude less than those in Table 9, since the vertex number is significantly reduced after we merge unambiguous k-mers into contigs. For example, the DBG of the HC-2 dataset has 46.97 M vertices, which is reduced to 1.0 M vertices after merging unambiguous k-mers into contigs, and further to 68,264 vertices after these contigs are merged after error correction.

6 Conclusion

We presented a scalable and flexible de novo genome assembler, PPA-assembler, built on a popular big data framework and provides strict performance guarantee. PPA-assembler is much faster than existing state-of-the-art distributed assemblers, and achieves comparable sequencing quality.

For future work, it is interesting to further consider paired-end information and to merge contigs into longer ones using scaffolding algorithms. However, we remark that our merging of k-mers into contigs is the bottleneck of scalability, and the data volume of the obtained contigs tends to be much smaller and can easily fit into a traditional scaffolding program.

Acknowledgments: The research of Guimu Guo and Da Yan is supported by NSF OAC 1755464 and NSF DGE 1723250. The research of Hongzhi Chen and James Cheng is supported by ITF 6904945 and GRF 14222816. The research of Jake Chen is supported by NIH/NCATS U54TR002731 and NCI/NIH/DHHS U01CA223976. The research of Zechen Chong is supported by NIMHD U54MD000502, NHGRI 3U01HG007301-06S1 and AHA 17IF33890015.

References

[1] A. Abu-Doleh and U. V. Catalyurek. Spaler: Spark and graphx based de novo genome assembler. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 1013–1018. IEEE, 2015.

[2] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski, et al. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology*, 19(5):455–477, 2012.

[3] S. Boisvert, F. Laviolette, and J. Corbeil. Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *Journal of Computational Biology*, 17(11):1519–1533, 2010.

[4] Y. Bu, V. R. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.

[5] J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, and D. S. Rokhsar. Meraculous: de novo genome assembly with short paired-end reads. *PloS one*, 6(8):e23501, 2011.

[6] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[7] T.-C. Chu, C.-H. Lu, T. Liu, G. C. Lee, W.-H. Li, and A. C.-C. Shih. Assembler for de novo assembly of large genomes. *Proceedings of the National Academy of Sciences*, 110(36):E3417–E3424, 2013.

[8] X. Feng, L. Chang, X. Lin, L. Qin, and W. Zhang. Computing connected components with linear communication cost in pregel-like systems. In *ICDE*, pages 85–96, 2016.

[9] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[10] A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler. Quast: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 2013.

[11] W. Huang, L. Li, J. R. Myers, and G. T. Marth. Art: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, 2012.

[12] D. Kleftogiannis, P. Kalnis, and V. B. Bajic. Comparing memory-efficient genome assemblers on stand-alone and cloud infrastructures. *PloS one*, 8(9):e75505, 2013.

[13] S. Liu and R. E. Tarjan. Simple concurrent labeling algorithms for connected components. In *SOSA@SODA*, pages 3:1–3:20, 2019.

[14] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3), 2015.

[15] R. Luo, B. Liu, Y. Xie, Z. Li, W. Huang, J. Yuan, G. He, Y. Chen, Q. Pan, Y. Liu, et al. Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. *Gigascience*, 1(1):18, 2012.

[16] I. MacCallum, D. Przybylski, S. Gnerre, J. Burton, I. Shlyakhter, A. Gnirke, J. Malek, K. McKernan, S. Ranade, T. P. Shea, et al. Allpaths 2: small genomes assembled accurately and with high continuity from short paired reads. *Genome biology*, 10(10):R103, 2009.

[17] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.

[18] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji. Swap-assembler: scalable and efficient genome assembly towards thousands of cores. *BMC bioinformatics*, 15(Suppl 9):S2, 2014.

[19] P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[20] S. Salihoglu and J. Widom. Gps: a graph processing system. In *SSDBM*, page 22, 2013.

[21] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, et al. Gage: A critical evaluation of genome assemblies and assembly algorithms. *Genome research*, 22(3):557–567, 2012.

[22] S. Sato. On implementing the push-relabel algorithm on top of pregel. *New Generation Comput.*, 36(4):419–449, 2018.

[23] Y. Shiloach and U. Vishkin. An o (logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[24] J. T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome research*, 22(3):549–556, 2012.

[25] 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 research*, 19(6):1117–1123, 2009.

[26] D. Yan, H. Chen, J. Cheng, Z. Cai, and B. Shao. Scalable de novo genome assembly using pregel. *CoRR*, abs/1801.04453, 2018.

[27] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.

[28] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 7(14):1821–1832, 2014.

[29] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *PVLDB*, 9(5):420–431, 2016.

[30] C. Ye, Z. S. Ma, C. H. Cannon, M. Pop, and W. Y. Douglas. Exploiting sparseness in de novo genome assembly. In *BMC bioinformatics*, volume 13, page S1. BioMed Central, 2012.

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[32] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

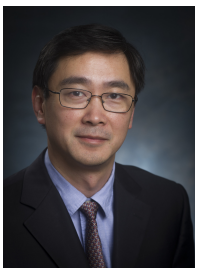
[33] A. V. Zimin, G. Marçais, D. Puiu, M. Roberts, S. L. Salzberg, and J. A. Yorke. The masurca genome assembler. *Bioinformatics*, 29(21):2669–2677, 2013.

Guimu Guo is a PhD student at the Department of Computer Science, the University of Alabama at Birmingham. His research interests include distributed computing systems, data science and bioinformatics.

Hongzhi Chen is a PhD student in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. He is interested in distributed computing systems and large-scale graph processing.

Da Yan is currently an Assistant Professor at the Department of Computer Science, the University of Alabama at Birmingham. He is the sole winner of Hong Kong 2015 Young Scientist Award in Physical/Mathematical Science. Dr. Yan regularly publishes in 1st-tier conferences and journals like SIGMOD, PVLDB, SIGKDD, ICDE, WWW, TKDE, TPDS, SoCC, EuroSys, PPoPP, etc. He also regularly serves as the reviewers of top journals including TODS, VLDBJ, TKDE, TPDS, etc., and serves in the program committees of top conferences such as SIGMOD 2019 and 2020, PVLDB 2018, IJCAI 2017, ICPP 2018, etc. Dr. Yan is the leading program co-chair of the BIOKDD 2018 and 2019 workshops held in conjunction with SIGKDD, and a guest editor of ACM/IEEE TCBB.

James Cheng is an Associate Professor with the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research focuses on big data infrastructures, distributed computing systems, and large-scale network analysis.



Jake Y. Chen is a Professor of Genetics, Computer Science, and Biomedical Engineering at the University of Alabama at Birmingham (UAB). He is also the Chief Bioinformatics Officer of UAB?s Informatics Institute and Head of the Informatics Section of the Genetics Department. He holds a BS degree in Biochemistry and Molecular Biology and MS and PhD degrees in Computer Science and Engineering. He has more than 20 years of research experience in biological data mining, systems biology, and translational bioinformatics, with more than 150 peer-reviewed publications. Prior to join UAB, he holds tenured faculty positions at Indiana University and Purdue University.

Zechen Chong is an Assistant Professor in the Department of Genetics and Informatics Institute at the University of Alabama at Birmingham. His group is focusing on development of algorithms for next-generation sequencing data and third generation sequencing data analysis. In particular, he is interested in novel algorithms to discover genomic rearrangements in health and disease genomes.