2018 IEEE 34th International Conference on Data Engineering

Scalable De Novo Genome Assembly Using Pregel

Da Yan[#], Hongzhi Chen[*], James Cheng[*], Zhenkun Cai[*], Bin Shao[†]

[#]*Department of Computer Science, The University of Alabama at Birmingham* yanda@uab.edu
[*]*Department of Computer Science and Engineering, CUHK* {hzchen, jcheng, zkcai}@cse.cuhk.edu.hk
[†]*Microsoft Research Asia* binshao@microsoft.com

***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. PPA-assembler adopts the popular *de Bruijn graph* based approach, and the operations run on Google's Pregel framework with strong performance guarantees. PPA-assembler demonstrates superior performance compared with existing assemblers, and is open-sourced at https://github.com/yaobaiwei/PPA-Assembler. The full version of this paper can be found at https://arxiv.org/abs/1801.04453.**

I. Introduction

Modern sequencing technologies generate many short DNA segments called *reads*, which are stitched together to generate longer DNA sequences for finding new genomes. Since single-threaded assemblers often require a high-end server with terabytes of RAM, many parallel/distributed assemblers have emerged [6], [1], [2], [4]. Instead of providing yet another parallel/distributed assembler, we abstract the key operations from existing assemblers, such as contig merging, tip removing and bubble filtering; we implement each operation as a distributed program with strong performance guarantees, and assemble them into a scalable distributed assembler called PPA-assembler. Users can also easily extend and reassemble the operations to implement various sequencing strategies.

PPA-assembler adopts the popular *de Bruijn graph* (DBG) based approach for sequencing [5]. We thus build it on top of our graph processing system Pregel+[1], which open-sources Google's Pregel framework for iterative graph computation. Each operation in sequencing is implemented as a distributed Pregel program where one iteration takes cost *linear* to the graph size, and the number of iterations is at most *logarithmic* to the graph size. Users can easily revise existing operations and write new operations using the intuitive "think like a vertex" programming model (to implement different sequencing strategies). Each operation may either read its input from Hadoop Distributed File System (HDFS), or directly obtain its input by converting the output of another operation in memory. Thus, the operations can be easily assembled, and PPA-assembler can readily inter-operate with existing Big Data systems by exchanging data through HDFS.

Paper Organization. Section II reviews Google's Pregel. Section III provides a brief introduction of DBG based assembly. Section IV presents the implementation of our various operations in PPA-assembler. Finally, we summarize the experimental results and conclude this paper in Section V.

[1]Pregel+: http://www.cse.cuhk.edu.hk/pregelplus/

II. Pregel Review

Given a graph $G = (V, E)$, we denote the number of vertices $|V|$ (resp. edges $|E|$) by n (resp. m). We also denote a vertex v's in-degree (resp. out-degree) by $d_{in}(v)$ (resp. $d_{out}(v)$). We abuse the notation and denote the ID of v also by v.

Pregel [3] distributes vertices to different machines in a cluster, where each vertex is stored with its adjacency list. 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* and update its value, 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 (e.g., vertex values) are dumped to HDFS.

We extend Pregel's API with the following two extensions, which are very useful in implementing PPA-assembler. Firstly, for two consecutive jobs j_a and j_b, we allow j_b to directly obtain input from the output of j_a in memory (no need to go through HDFS). Users need to define a user-defined function (UDF) *convert*(v) which indicates how to transform an object v of job j_a's vertex class to job j_b's vertex class. Since Pregel+ distributes vertices to machines by hashing vertex ID, the converted vertex objects are shuffled according to their new ID. Secondly, the input data may not be in the format of one line per vertex. For example, each line may correspond to one edge. To create vertices, a mini-MapReduce procedure is supported during graph loading to group edges by source vertex ID, so that each group can be reduced into a vertex object v along with an adjacency list of v's out-neighbors.

Our prior work [7] defined a class of scalable Pregel algorithms called PPAs (practical Pregel algorithms), and it designed PPAs for many fundamental graph problems which can be used as building blocks of other problems. Formally, a Pregel algorithm is called a *balanced PPA* (BPPA) if it has (1) *linear space usage:* each vertex v uses $O(d_{in}(v) + d_{out}(v))$ space of storage; (2) *linear computation cost:* the time complexity of $v.compute(.)$ is $O(d_{in}(v) + d_{out}(v))$; (3) *linear communication cost:* at each superstep, the volume of the

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00114

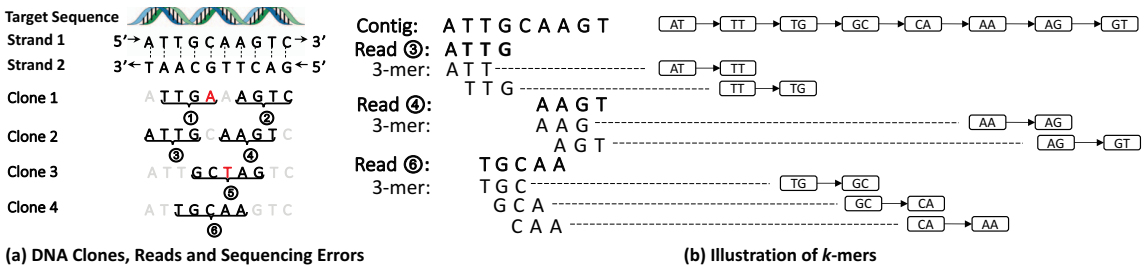

Figure 1. Reads and k-mers

messages sent/received by each vertex v is $O(d_{in}(v)+d_{out}(v))$; (4) the algorithm terminates after $O(\log n)$ supersteps.

Constraints (1)–(3) offer good load balancing, but for some problems we can only achieve *overall linear space usage, computation and communication cost*. Such a Pregel algorithms is simply called a PPA.

III. De Novo Genome Assembly

This section provides a brief background on DBG-based sequencing; a complete review can be found at the full version.

We model a DNA molecule as a very long sequence of nucleotides, where each nucleotide can take one of the four base types A, C, G and T. Since sequencing long DNA segments is error-prone, modern sequencing technologies generate a large number of short DNA segments, called *reads*. Figure 1(a) illustrates this process, where 4 DNA clones are sheared into 6 reads. Note that a DNA molecule consists of two strands coiled around each other, and we only consider strand 1 in Figure 1(a) for simplicity. Reads can have variable lengths, and sequencing errors may happen at some positions such as in reads ① and ⑤ (errors highlighted in red). Also, reads may overlap with each other, such as reads ② and ④ that share the segment "AGT". It is through these overlaps that genome assembly algorithms stitch reads to get longer sequences (called *contigs*).

The DBG-based assembly approach first constructs a de Bruijn graph (DBG) from the reads, and then finds contigs from the DBG. To construct a DBG, each read is cut into consecutive sub-sequences of length $k+1$, where each sub-sequence is called a $(k+1)$-mer. For example, Figure 1(b) illustrates how we can generate 3-mers from reads ③, ④ and ⑥ of Figure 1(a) (here $k=2$), where read ③ "ATTG" can be cut into two 3-mers "ATT" and "TTG". For each $(k+1)$-mer, we define its *prefix* (resp. *suffix*) as the subsequence without the last (resp. first) nucleotide, which is a k-mer.

The k-mers define the vertices in the DBG, and each $(k+1)$-mer defines an edge from its prefix to its suffix in the DBG. For example, in Figure 1(b), the first 3-mer of read ③, i.e., "ATT", defines a directed edge in DBG from vertex "AT" to vertex "TT". All the 3-mers in Figure 1(a) create a path as shown on the top right of Figure 1(b), which stitches reads ③, ④ and ⑥ together into a longer contig "ATTGCAAGT".

Ideally, if k is large enough, any sub-sequence of length k in the whole DNA sequence appears only once, i.e., any k-mer vertex of the DBG corresponds to a unique sub-sequence in the whole sequence. In this case, the DBG is essentially a path following which we can reconstruct the whole sequence. However, k cannot be arbitrarily large in practice, since reads are short and any read with length less than $(k+1)$ will be ignored. A k-mer vertex is ambiguous if it corresponds to several different segment positions in the whole sequence. While the whole sequence corresponds to a Eulerian path of the DBG, there can be many Eulerian paths and our goal is to find the maximal simple paths in the DBG that do not contain any ambiguous vertex, which constitute contigs.

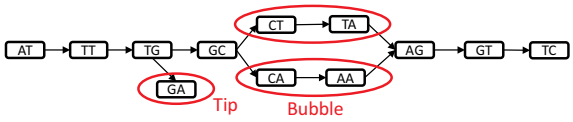

Figure 2. De Bruijn Graph

Read errors can further complicate the assembly process by introducing false vertices and edges into the DBG. Two typical errors are *tips* and *bubbles*, as illustrated in Figure 2 which shows the DBG constructed from the reads of Figure 1(a). A *tip* is a short dangling path in the DBG that leads to a dead-end, such as edge "TG"→"GA" in Figure 2 contributed by the error in read ①. A *bubble* is a sub-path that starts from a certain vertex at the main path of the DBG, and returns to the same path after a few hops. Figure 2 shows a bubble where the main path "GC"→"CA"→"AA"→"AG" is contributed by correct reads such as ④ and ⑥, and the erroneous sub-path "GC"→"CT"→"TA"→"AG" is caused by erroneous read ⑤.

If we can correct the errors, we can obtain longer contigs, but overly aggressive strategies may lead to false alarms that create wrong (albeit longer) contigs. For example, a long tip needs to be generated by multiple errors which is unlikely. For bubbles, we remove sub-path(s) with a very low coverage. Here, the *coverage* of an edge is defined as the number of reads that generate it. A correct path is unlikely to have a low coverage as there are many DNA clones. We also require a sub-path to be similar to the main path (with high coverage) in order to remove it, since it is unlikely to have multiple errors that significantly changes the corresponding sub-sequence.

So far, we only discussed the case of one strand. In reality, reads may be obtained from both strands of the DNA molecule. In this case, additional manipulations on edge directionality (e.g., reverse complement) is needed to account for the chemical orientation of each strand. Due to space limit, we refer readers to our full version for the orientation-related details.

IV. PPA-Assembler Algorithms

We first present our compact graph data structures, and then describe the operations supported by PPA-Assembler.

A. Vertex & Edge Formats

Since genome assembly has a very high memory demand, we design compact data structures for vertices and edges. Due to space limit, we only describe them briefly and leave the complete format information to the full version of this paper.

Each vertex in a Pregel program has a unique ID for message passing, and we use integer to specify vertex ID for efficiency. There are two kinds of vertices: (1) k-mer and (2) contig. We encode the sequence of a k-mer directly into its integer ID. Recall that reads are cut into $(k+1)$-mers during DBG construction; assume that $k \leq 31$, then we use 64-bit integer for ID as follows (if $k > 31$, we use a big integer with more bits). Each nucleotide is represented by two bits: A (00), T (11), G (10), C (01), and thus a k-mer requires at most 62 bits to represent. The first 2 bits can be flipped to 1 to indicate that a k-mer or a contig has no neighbor along one direction (e.g., the dead-end of a tip). In contrast, since a contig can be an arbitrarily long sequence, we cannot encode the sequence into the contig's ID. Instead, since the contigs are distributed among the machines after their generation, we let the i-th worker machine assign its j-th contig a 64-bit ID that equals the 32-bit integer representation of i (with the first 2 bits fixed to 00) concatenated with the 32-bit integer representation of j. To avoid collision with the ID of a k-mer, we also flip the most significant bit to be 1.

Each vertex also maintains an adjacency list of its neighboring vertices (k-mers or contigs). Since a contig is obtained by merging unambiguous k-mers, it has only two neighbors along its two opposite sequencing directions, where each neighbor is either an ambiguous k-mer or the dead-end. In contrast, a k-mer vertex may have up to 4 k-mer in-neighbors and 4 k-mer out-neighbors (i.e., connecting A/T/G/C on either side), and we compress its adjacency lists using compact bitmaps to save memory space. A k-mer vertex tracks its contig neighbors differently from the k-mer neighbors (to be described shortly).

A contig vertex keeps its sequence as a variable-length bitmap, and also maintains its own coverage, which is computed as the minimum coverage of all edges (i.e., $(k+1)$-mers) merged by the contig. A k-mer vertex v tracks a contig neighbor u by also maintaining u's other k-mer neighbor (denoted by w) for direct message passing, and u is essentially edge (v,w) that can keep information like sequence length and coverage to facilitate tip removing and bubble filtering.

A k-mer vertex can be of one of three types: (a) $\langle 1 \rangle$: such a vertex only has one neighbor, and is thus a dead-end; (b) $\langle$1-1$\rangle$: such a vertex has one in-neighbor and one out-neighbor and is thus unambiguous; (c) $\langle m\text{-}n \rangle$: such a vertex has at least two neighbors but is not of type $\langle$1-1$\rangle$, and it is ambiguous. Note that k-mer vertex is contributed by the prefix or suffix of a $(k+1)$-mer and thus must have at least one neighbor.

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$. It is possible to have an isolated contig where both ends are dead-ends, and we treat it as of type $\langle 1 \rangle$ (i.e., a tip).

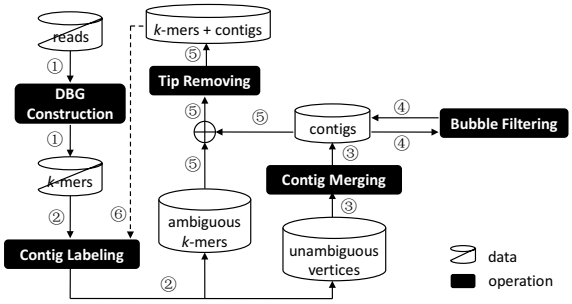

Figure 3. Operation Diagram

B. Operations and Their Algorithms

PPA-assembler provides a library of operations, and each is a PPA. Figure 3 shows the data flow diagram of PPA-assembler, which includes five operations: ① *DBG construction*, which constructs a DBG from the DNA reads; 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 with a similar sequence; ⑤ *tip removing*, which takes the ambiguous k-mers and the contigs (after bubble filtering), and removes tips. The output of tip removing can be fed to the "contig labeling" operation again to grow longer contigs (see arrow ⑥), since the previous error-correction operations may have converted some ambiguous k-mer vertices into unambiguous ones, and operations ②–⑤ may loop as needed.

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, which form the DBG. In Phase (i), *map(.)* parses each read into $(k+1)$-mers using a sliding window of $(k+1)$ elements (c.f. Figure 1(b)). The sequence of a $(k+1)$-mer is directly encoded in its 64-bit integer ID, which functions as the key for shuffling. Each machine aggregates the count of each generated $(k+1)$-mer, and the "$(k+1)$-mer, count" pairs are shuffled for *reduce(.)* to obtain the total count of each $(k+1)$-mer. A $(k+1)$-mer is filtered if its coverage is no more than a user-defined frequency threshold, since this $(k+1)$-mer is very likely to be contributed by erroneous readers.

In Phase (ii), *map(.)* extracts two k-mers from each non-filtered $(k+1)$-mers. If an extracted k-mer is newly obtained on a machine, the machine 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 their adjacency lists. Edge count (equal to the $(k+1)$-mer's count) is also recorded or incremented. The k-mer vertices with partially constructed adjacency lists are then shuffled by the 64-bit integer ID, and

reduce(.) merges the partially constructed adjacency lists of each k-mer into a complete one (including the edge coverage).

② **Contig Labeling.** Let us call a path that only contains vertices of types $\langle 1 \rangle$ and $\langle 1\text{-}1 \rangle$ 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. A vertex is at one end of a maximal unambiguous paths, if its type is $\langle 1 \rangle$, or if its type is $\langle 1\text{-}1 \rangle$ and at least one neighbor is of type $\langle m\text{-}n \rangle$. The operation first recognizes contig-ends in two supersteps: (1) in superstep 1, every $\langle m\text{-}n \rangle$ vertex broadcasts its ID to all its neighbors, and then votes to halt; it will never be reactivated again as subsequent computation only involves unambiguous vertices; (2) in 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.

We first let contig-end vertices remove all their edges with ambiguous vertices, so that the DBG graph becomes a set of isolated unambiguous paths, each corresponding to a contig.

To find all contigs in $O(\log n)$ supersteps, we run a variant of the BPPA for list ranking proposed in [7]. In this algorithm, each unambiguous vertex maintains a pair of IDs, which is initialized as the pair of neighbor IDs. Then, pointer jumping [7] is performed in both directions of every unambiguous path in parallel, till every unambiguous vertex v obtains the pair of IDs of two contig-end vertices for v's path. This is the key algorithm which also handles the orientation related issues, and the complete algorithm is given in our full paper version.

Bidirectional list ranking alone is not sufficient if the DBG contains a cycle of vertices of type $\langle 1\text{-}1 \rangle$, since these vertices will never reach an end. Therefore, if the number of active vertices is larger than 0 and does not decrease after a round, the PPA for finding connected components proposed in [7] is run on the remaining active vertices, so that each vertex in a cycle obtains the smallest ID in the cycle. We actually run an improved algorithm which is given in our full paper version.

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.

Vertices are stitched from a contig-end vertex v_{first}, which contains a neighbor not in the group; if such a vertex cannot be found, the contig is cycled and we start stitching from an arbitrary vertex. If the last vertex v_{last} to stitch is of type $\langle 1 \rangle$, we exit *reduce*(.) if the aggregated contig length is not above the user-specified tip-length threshold. In all other cases, the stitched contig is output with its coverage set as the minimum edge coverage seen during the concatenation, and with its two neighbors set as v_{first}'s in-neighbor and v_{last}'s out-neighbor.

④ **Bubble Filtering.** The constructed contigs may be further filtered with "bubble filtering" through 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*(.), which prunes low-coverage contigs whose edit distance to an unpruned contig is within a user-defined threshold.

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. This takes 2 supersteps: (i) in superstep 1, each contig vertex sends its content (including length) to both neighbors (if not contig-ends); 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, and length of adjacent contigs is already collected by every k-mer vertex, only k-mer vertices need to participate in the operations. Note that the tip removing may change some $\langle m\text{-}n \rangle$ vertices into $\langle 1 \rangle$ vertices, hence generating new tips. We thus run multiple phases of vertex-centric tip removing till no new $\langle 1 \rangle$ vertex is generated.

In a phase, request-messages start passing from each $\langle 1 \rangle$ vertex, where cumulative sequence length is recorded. A request-message received from one neighbor of a vertex v (which is $\langle 1\text{-}1 \rangle$-typed) is relayed to the other neighbor of v with cumulative sequence length updated, till a vertex of type $\langle m\text{-}n \rangle$ or $\langle 1 \rangle$ is reached. The vertex checks whether the cumulative sequence length is not larger than the tip-length threshold, and if so, it sends a delete-message back. The delete-message is relayed through $\langle 1\text{-}1 \rangle$ vertices till reaching the $\langle 1 \rangle$ vertex that initiates the request-message, and vertices and contigs along the relaying path delete themselves.

V. Experiments & Conclusions

We compared with existing parallel assemblers in both efficiency and result quality, and the complete experiments are reported in our full paper version. Our findings are that PPA-assemblers is many times faster than other distributed assemblers, and achieves comparable sequencing quality.

Acknowledgments. The work was partially supported by CUHK 14206715 & 14222816 from Hong Kong RGC, NSF DGE-1723250, Microsoft Azure Research Award, and MSRA grant 6904224.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.