Efficient Out-of-Core Contig Generation

Julio Omar Prieto Entenza, Edward Hermann Haeusler, and Sérgio Lifschitz(✉)

Departamento de Informática - (PUC-Rio), Rio de Janeiro, Brazil
sergio@inf.puc-rio.br

Abstract. Genome sequencing involves splitting a genome into a set *reads* that are assembled into *contigs* that are eventually ordered and organized as *scaffolds*. There are many programs that consider the use of the *de Bruijn* Graph (dBG) but they must deal with a high computational cost, mainly due to internal RAM consumption. We propose to use an external memory approach to deal with the *de Bruijn* graph construction focusing on contig generation. Our proposed algorithms are based on well-known I/O efficient methods that identify unitigs and remove errors such as tips and bubbles. Our analytical evaluation shows that it becomes feasible to generate *de Bruijn* graphs to obtain the needed contigs, independently of the available memory.

1 Introduction

Genome sequencing is the process that determines the order of nucleotides within a DNA molecule. Modern instruments splits a genome into a set of many short sequences (*reads*) that are assembled into longer contiguous sequences, *contigs*, followed by the process of correctly ordering *contigs* into *scaffolds* [18].

We may associate *genome sequencing* with the problem of finding a Hamiltonian Cycle through an Overlap Layout Consensus (OLC) assembly method. Alternatively, it can be modeled as the problem of finding a Eulerian Cycle considering the *de Bruijn Graph* (dBG) based methods [14]. The latter can be seen as a breakthrough for the research on genome assembly. This is due to the fact that to find a Hamiltonian Cycle is an NP-complete problem [14].

When we handle actual dBGs, we must consider the existence of errors that appear due to high-frequencies distortions on Next-generation Sequencing (NGS) platforms. These errors induce the dBG size to be more prominent than the *overlap* graph used in the OLC genome sequencing method using the same reads.

To remove the errors, we need to have some data structure representation of the dBG. Current real-world datasets induce challenging problems as they have already reached high volumes and will continue to grow as sequencing technologies improve [19]. As a consequence, the dBG increases the complexity leading to tangles and topological structures that are difficult to resolve [16]. Also, the graph has a high memory footprint for large organisms (*e.g.,* sugarcane plants) and it becomes worse due to the increase of the genome datasets. Therefore, there are research works that focus on dealing with the ever-growing graph sizes for dBG-based genome assembly [3,15]. Any proposed solution must be aware of the fact that the dBG is not entirely built before error pruning. After the splitting of the genome in the first phase and, for a natural number k chosen based on an empirical criterion, for each *read* of size m we will have $m-k+1$ possible k-mers, which correspond to the nodes of the dBG. The number of k-mers depends on the adjacency determined by the $r$ead itself. The splitting error introduced in a given *read* may large increase the size of the subgraph it induces.

Roughly speaking, the dBG is a set of k-mers (subsequences of length k) [23] linked to each other, according to information provided by their *reads*. Some k-mers can come from different *reads* and the information about adjacency supplied by any other *read* is processed only at dBG constructing phase. Thus, error pruning should happen at this particular stage. The memory needed is so significant that we must use external memory to accomplish the dBG construction.

The different approaches that deal with the dBG size aim to design lightweight data structures to reduce the memory requirements and to fit the assembly graph into the main memory. Although it might be efficient, the amount of memory increases according to the size of the dataset and the DNA of the organism. While bacteria genomes currently take only a few gigabytes of RAM, species like mammalian and plants require over tens to hundreds of gigabytes. For instance, approximately 0.5 TB of memory is required to build a hash table of 75-mers for the human genome [19].

We propose algorithms to simplify and remove errors in the *de Bruijn* graph using external memory. As a result, it will be able to generate contigs using a fixed amount of RAM, independently of the read dataset size. There are other works addressing *de Bruijn* graph processing using external memory [8,11], but they focus only on the constructing of large *de Bruijn* graphs efficiently with no error prune considerations. To the best of our knowledge, this is the first proposal of using an external memory approach focusing on the dBG simplification and errors removal for contig generation during dBG construction.

We show an algorithm that provides out-of-core *contraction of unambiguous paths* with an I/O cost of $O(|E|/B)$, where E is the set of edges of the dBG and B is the size of the partition loaded to the RAM each time. With the overhead for creating the new partitions, the overall I/O complexity is $O((sort(|E|) + |E|/B)\log Path)$, where $Path$ is the length of the longest unambiguous path in the dBG. For a machine with memory M, and a dBG satisfying $|E| < M^2/4B$ $sort(E)$ is performed with I/O complexity $O(|E|/B)$ [9]. Summing up the I/O complexity of this out-of-core contraction of paths is $O((|E|/B)\log Path)$. The out-of-core *graph cleaning* phase, by removing *tips* and *bubbles*, is performed with a similar I/O complexity $O((|E|/B + sort(|E|))logPath)$. The *creation of contigs* is performed by a full scan of the graph with I/O complexity $O(|V|/B)$.

2 De Novo Assembly Using *de Bruijn* Graph

Given a *de Bruijn* Graph $G = (V, E)$ for genomic sequence assembly each node holds unique k-mers, and the edges reflect consecutive nodes if they overlap by

$k-1$ characters. The assembly aims to construct a set of contigs from the dBG G. Given a dBG as input, to generate contigs is equivalent to output all contiguous sequences which represent unambiguous paths in the graph.

The use of the dBG to generate contigs consists of a pipeline: nodes enumeration, compaction, and graph cleaning. In the first step, a set of distinct k-length substrings (*k-mers*) is extracted from the reads. Each *k-mer* becomes a graph node. Next, all paths with all but the first vertex having in-degree 1 and all but the last vertex having out-degree 1 (*unitigs*) are compacted into a single vertex. Finally, the last step removes topological issues from the graph due to sequencing errors and polymorphism [5].

The number of nodes in the graph can be huge. For instance, the size of the genome of white spruce is 20 Gbp and generates 10.7×10^9 *k-mers* (with k = 31) and needs 4.3 TB of memory [5]. Also, the whole genome assembly of 22 Gbp (bp - base pairs) loblolly pine generates 13×10^9 k-mers and requires 800GB of memory [5]. Theoretically speaking, a 1,000 Genomes dataset with 200 Terabytes of data can generate about 2^{47} or nodes, 64–128 times larger than the problem size of the top result in the Graph 500 list [15].

Next-generation sequencing platforms do not provide comprehensive read data from the genome sequences. Hence, the produced data is distorted by high frequencies of sequencing errors and genomic repeats [18]. Sequencing errors compound this problem because each such error corrupts the correct genomic sequence into up to k erroneous k-mers. These erroneous k-mers introduce new vertices and edges to the graph, significantly expanding its size and creating topological artifacts as tips and bubbles [23].

Different solutions have been proposed to address the memory issues in genome assembly problem. One approach samples the entire k-mer set and performs the assembly process over the selected k-mers [21]. Another approach address to encode the dBG into efficient data structures such as light-weight hash tables [3], succinct data structures [2] or bloom filters[6,17]. There are research works based on distributed memory systems for processing power and memory demanding resources [3,7,15].

Although their apparent differences, all of these approaches are based exclusively on in-memory systems. Consequently, if the size of the graph exceeds the amount of memory available, it will be necessary to increase the RAM. As in the next future, the size of datasets will increase dramatically, and this situation will stress the different systems [20]. There is a need for new approaches to process all of this massive amount of information in a scalable way. We propose in this work to use an external memory approach to process the dBG. To increase the amount of RAM does not guarantee that the graph will always fit.

3 Overview of Our Proposed Approach

Our basic pipeline of *de novo* genome assembly could be divided into five basic operations [23]: 1) *dBG construction*, which constructs a dBG from the DNA reads; 2) *Contraction of unambiguous paths*, which merges unambiguous vertices into unitigs; 3) *Graph cleaning*, which filters and removes errors such as tips and bubbles; 4) *Contigs creation*, which create a first draft of the contigs and 5) *Scaffolds*, which joins the previous contigs together to form longer contigs. In this work, we face steps 2, 3, and 4 using an external memory approach. We assume a *de Bruijn* graph exists, and it is persisted as an edge-list format.

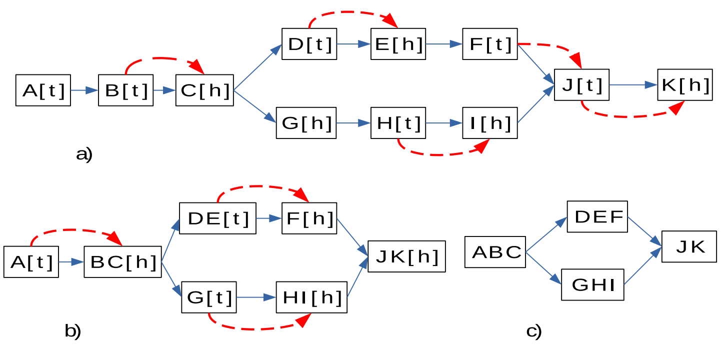

Fig. 1. Graph Contraction. Dashed arcs represent the messages and the label between brackets indicates if a vertex is a tail (t) or a head (h). a) A flipped coin choose which node will be t or h. If a tail vertex has out-degree = 1 then it sends a message to its neighbour. b) If a h vertex receives a message and its in-degree = 1 then both vertices are merged. c) Shows the result after some repeated steps.

The **Contraction of unambiguous paths** simplifies the graph by merging some nodes in the graph. Whenever a node with only one outgoing directed edge points out to another node with only one incoming directed edge, these two nodes are merged. These single nodes are called *unitigs* and are maximal if they extend in either direction. Thus, the problem of compacting a *de Bruijn* graph is to report the set of all maximal *unitigs*. Figure 1 shows how we simplify the dBG into a compacted graph. All the remained nodes are maximal *unitigs*.

The algorithmic solution to this problem is straightforward in the in-memory context. Let's assume that each path representing a unitig is a linked list where the head and the tail can be branching nodes (see Fig. 1a). Then, to obtain a maximal unitig, we only need to visit each u_c node and merge them with its successor node into the new one.

Although the in-memory algorithm is straightforward, the use of this approach is not efficient in the external memory because the number of I/O accesses is linear concerning the number of nodes. Finding all the maximal unitigs is analogous to apply the well-known graph edge contraction technique in external memory on all the unambiguous paths. Given a graph G, the contraction of an edge (u, v) is the replacement of u and v with a single vertex such that edges incident to the new vertex are the edges other than (u, v) that were incident with u or v [22]. As in our case, this algorithm can only apply on unambiguous paths. Therefore, we divided our proposal into two steps: 1) select branching nodes u_a

as head/tail of each unitig paths; and 2) apply the graph contraction technique over these paths until all nodes are maximal unitigs.

In *Graph cleaning* we aim to remove short dead-end divergences, called **tips**, from the main path. One strategy consists of testing each branching node for all possible path extensions up to a specified minimum length. If the length of the path is less than a certain threshold (set by the user) then the nodes belonging to this path are removed from the graph [7,23].

The tips removing process is analogous to traversal the paths from a branching node, u_a, to a dead-end node u_e. The graph does not fit into RAM, even after the unitig process. We need to traverse the dBG in an I/O efficient way to find and remove all tips. Our algorithm is based on an external memory list ranging from the u_e to u_a nodes. However, we have to make two significant modifications: (i) the ranking is represented by each edge/node's coverage to decide which path will be removed, and (ii) as two of more dead-ends could reach the same u_a node, we need to keep in RAM a data structure to make the traversal backward. This way, we eliminate the selected path from the branching node.

Bubbles are paths that diverge from a node then converge into another. The process of fixing bubbles begins by detecting the divergence points in the graph [23]. For each point, all paths from it are detected by tracing the graph forward until a convergence point is reached. Some assemblers restricts the size of the bubble to n nodes where $k \leq n \leq 2k$ [7], others use a modified version of Dijkstra's algorithm [23]. To simulate the different external memory approaches, we need to identify all branching nodes u_a. We execute an I/O-efficient breath-first search (BFS) from u_a until we find a visited node. It means that there is a bubble at some point in the search (v_b). Then, we select the branch that will be kept and start another BFS in a backward direction (the start node is v_b). Finally, we remove the other paths until we find back u_a. After the execution of steps 2 and 3, the *Contigs creation* step involves the output of all the contigs represented by nodes.

Processing Out-of-Core Graphs. Many graph engines implement a vertex-centric or "think like a vertex" (TLAV) programming interface. This paradigm iteratively executes a user-defined program over vertices of a graph. The vertex program is designed from the vertex's perspective, receiving as input the data from adjacent vertices and incident edges. Execution halts after a specified number of iterations, called supersteps, are completed. It is important to note that each vertex knows the global supersteps. There is no other knowledge about the overall graph structure but its immediate neighbors and the messages that are exchanged along their incident edges [13].

The computation graph engines proceed in supersteps. For every superstep, it loads one or more partitions p based on available RAM. Then, it processes the vertices and edges that belong to p and saves the partitions back to disk. A different subset of partitions is then loaded and processed until all of them have been treated for the given superstep. The process is then repeated for the next superstep until there are no remaining vertices to visit (see Algorithm 1

Algorithm 1. Out-of-core graph processing taken from GraphChi [9]

```
Input:
    $G = (V, E)$: a general graph
    UpdateFunction: a user-defined program to apply over the vertices
1: procedure OutOfCore($G$, $UpdateFunction$)
2:     for each superstep ∈ Algorithm do          ▷ superstep is a global variable
3:         for each p ∈ Partition do
4:             p.load()                            ▷ Load partition to RAM
5:             parallel for each v ∈ p.vertices() do   ▷ Apply UpdateFunction
6:                 UpdateFunction($G$,$v$)
7:             p.save()                            ▷ Save partition to disk
```

from GraphChi [9]). If the machine has sufficient RAM to store the graph and metadata, all partitions can be kept in RAM, and there is no disk access [9].

Because *all the operations related to partitions and parallel vertices processing are fixed*, from now we will only highlight the *UpdateFunction* and the number of supersteps. For simplicity, *UpdateFunction(G,u)* means that we apply the function over the vertex u in the graph G.

4 Contig Generation

The graph contraction algorithm (Algorithm 2) is based on I/O-efficient list ranking algorithm based on graph contraction [4] but using a TLAV [12]. The output is the distance of each node from the beginning of the list. In this case, it corresponds to k-mer concatenation. Thus, in our output, the beginning node will have the unitig concatenation. Initially, we assign the k-mer as the rank of each vertex. We then continue recursively: first, in one superstep, we find a maximal independent set among all vertices that belong to a unitig (line 11). For each of these, we flip a fair coin and vertices that flipped "tails" pick a neighbor that flipped "heads" (if any) to contract with [1]. Later the vertices identified as "tails" with precisely one outgoing edge send the required information to its neighbors (lines 12 and 13). In the next superstep, the vertices marked as head, with only one in-going edge, are merged with tail vertices, and the edge information is updated (lines 17 to 22). The function *PreprocessNewGraph* creates new partitions from the removed and added vertices and edges (line 9). This step implies merging nodes, and remove duplicate edges and update the graph partitioning. Next, we recursively continue coarsening the graph until all unambiguous nodes are collapsed.

I/O Analysis. Lines 11–22 can be done with a full scan over all the graph partitions. We load a partition into RAM, we update the vertices and edge values, and then write them to the disk. Then, we load the next partition and its vertices and edges according to the saved partition. So, the I/O cost is $O(|E|/B)$.

On the other hand, *PreprocessNewGraph* (line 9) creates new partitions from superstep $i - 1$ to be processed in the superstep i. To create the new

Algorithm 2. Graph Contraction

```
Input: $G = (V, E)$: a contracted dBG
Output: $G'$: a graph with all no ambiguous paths contracted.
1: procedure ContractGraph($G$)
2:     $superstep \leftarrow 0$ and $merge \leftarrow true$
3:     while $merge = true$ do
4:         $merge \leftarrow false$
5:         $MergeNodes(G, u)$                          ▷ External Memory Context
6:         $superstep \leftarrow superstep + 1$
7:         if $superstep$ is odd then
8:             $G \leftarrow PreprocessNewGraph(G)$
9: procedure MergeNodes($u$)
10:    $flag \leftarrow T \vee H$ in randomly way
11:    if superstep is even then
12:        if $flag = true \wedge d^{+}(u) = 1$ then
13:            $send(\{flag, seq, neighbors, id\}, j)$     ▷ message to outgoing edge
14:    else
15:        if $d^{-}(u) = 1$ then                  ▷ merge nodes from incoming neighbor
16:            $m \leftarrow receive(\{flag, seq, i\})$
17:            if $m.flag \neq flag$ then
18:                $add\_edge(u, m.neighbors)$
19:                $seq \leftarrow glue(seq, m.seq)$
20:                $delete(m.id)$
21:                $merge \leftarrow true$
```

partitions, first, it is necessary to divide the nodes by their ID and later sort all the edges based on their destination vertex ID [9]. Thus, the I/O cost of the process for the created graph is $O(sort(|E|))$.

Finally, at each superstep, Algorithm 2 contracts a constant fraction of the vertices per iteration by graph contraction [22]. It expected $O(\log Path)$ iterations with high probability, where $Path$ is the longest unambiguous path in the graph [1]. Hence, the overall expected I/O cost for simplifying the graph is $O((sort(|E|) + |E|/B) \log Path)$. If we assume $|E| < M^2/(4B)$ then $sort(|E|) = O(|E|/B)$. This condition can be satisfied with a typical value of M, say 8 GB, B in the order of kilobytes and a graph size smaller than a petabyte [10]. On these conditions the I/O cost is $O((|E|/B) \log Path)$ (Fig. 2).

To **remove tips**, we design a straightforward procedure in few supersteps (Algorithm 3). At this point, all nodes are contracted. Thus, all terminal nodes are potential tips, and they may be removed. In the first superstep, the vertices having an in-degree or out-degree of zero and sequence's length less than $2k$ are identified as potential tips (line 10), and a message is sent to their neighbors (line 11). In the next superstep (lines 14–17), the vertices that received the messages, search for the maximal multiplicity among all neighbors and remove those potential tips with multiplicity less than the maximal value. Removing the tips generates new linear paths in the graph that will be contracted. Note

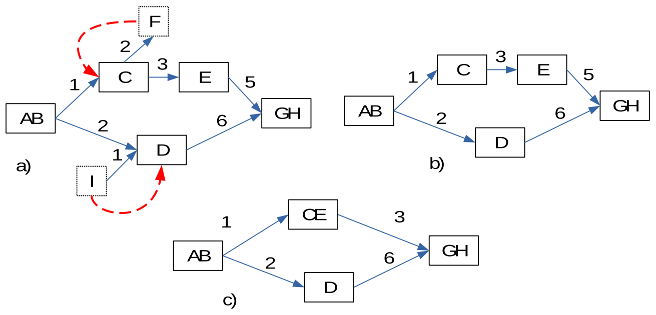

Fig. 2. Removing tips. Dotted lines show the sent messages. a) F and I are marked as potential tips. Later, they send a message to their neighbors. b) C and D receive the messages and check their multiplicity to eliminate the real tips. So, H and I are removed. c) The graph is compressed to obtain the final result.

that once the initial set of tips are removed, it could produce other tips. Most assemblers execute the removal tip algorithm a fixed number of times.

Algorithm 3. Tips removal

```
Input: $G = (V, E)$: a dBG
Output: $G'$: another graph with tips removed.
 1: procedure TIPS($G$)
 2:     $superstep \leftarrow 0$
 3:     while $tips = true$ do
 4:         $tips \leftarrow false$
 5:         $RemoveTips(G, u)$
 6:         $superstep \leftarrow superstep + 1$
 7:     $ContractGraph(G))$
 8: procedure REMOVETIPS($u$)                          ▷ External Memory Context
 9:     if $superstep = 0$ then
10:         if u is $u_e \wedge |seq| \leq 2k$ then          ▷ Identify all potential tips
11:             $send(id)$                             ▷ Send a message to its neighbor
12:     else
13:         $maximal \leftarrow \max(\forall u.neighbors.multip)$     ▷ Get max multiplicity
14:         for all $m \in receive(id)$ do ▷ Identify the real tips and remove them
15:             if $(u, m.id).mult < maximal$ then
16:                 $delete\_node(m.id)$
17:                 $tips \leftarrow true$
```

I/O Analysis. The function *RemoveTips* needs only two supersteps to remove any tip given a graph: one to identify all tips and other to removes all of them (lines 8–17). This can be carried out with at most two full scans over all the graph partitions. Thus, the I/O cost is $O(|E|/B)$. Although *RemoveTips* only executes two supersteps, the graph contraction dominates the I/O cost (line 7). The I/O cost is $O((|E|/B + sort(|E|)) \log Path)$, where *Path* represents the longest path created after the tips are removed.

The primary approach to **identify and remove bubbles** is based on BFS (breadth-first search) algorithm. As bubbles consist of paths with very different multiplicities, those paths with low multiplicity are deleted and use the path with the highest multiplicity to represent the bubble. In our algorithm, each vertex manages its history, which makes it easy to control the different paths and to pick up the right one.

The proposed algorithm has two stages: forward and backward. In the forward stage (Algorithm 4) identifies all paths that form a bubble and select one of them. On the other hand, the backward stage, (Algorithm 5), removes the redundant paths and compacts the graph. Due to space limitations, we will illustrate and explain both algorithms by examples. See Fig. 3 for the forward stage and Fig. 4 for the backward stage.

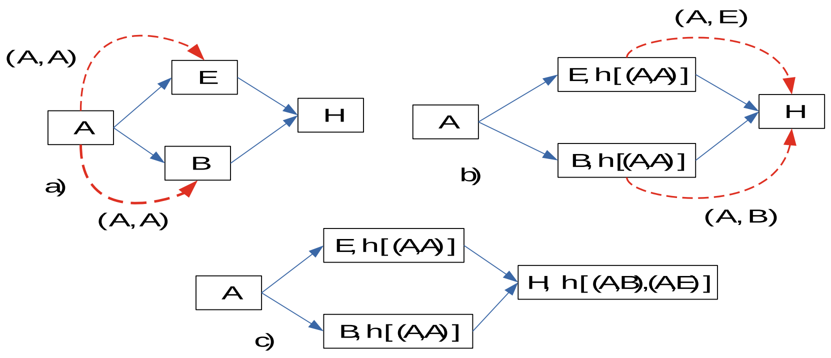

Fig. 3. Forward Bubble Detection. The figures only show the id par in the sent messages. a) A is a potential bubble beginning, so it sends messages to the outgoing neighbors. b) B and E keep it and forward new messages updating the vertices IDs. c) When H, in-degree ≥ 2, receives a message it selects that E belongs to a bubble. Later, it marks the node for the backward stage.

I/O Analysis. In the forward stage, *SelectPath* (line 4) iterates through the bubbles in a constant number of times depend on a *limit* value. Also, only a small number of messages are present in the graph, each one originating from any ambiguous vertex whose out-degree is greater than 2. Moreover, each message will be passed along an edge exactly once, as notifications are only sent

Algorithm 4. Forward stage bubble detection

```
Input:
    $G = (V, E)$: a compressed dBG and $limit$: max. length of the bubble path
Output: $G'$: a graph with all bubbles selected.
 1: procedure FindBuble($G$)
 2:     $superstep \leftarrow 0$
 3:     while $superstep \leq limit$ do
 4:         $SelectPath(G, u))$
 5:         $superstep \leftarrow superstep + 1$
 6: procedure SelectPath($G$,$u$)                          ▷ External Memory Context
 7:     if $superstep = 0 \wedge d^{+}(vertex) \geq 2$ then ▷ Identify possible bubble start
 8:         $outgoing.add(id, id, seq)$
 9:     else
10:         $outgoing \leftarrow \{\}$ and $incomming \leftarrow receive(id1, id2, seq)$
11:         for each $m \in incomming$ do
12:             if $m.id \notin history$ then
13:                 $m.seq \leftarrow glue(seq, m.seq)$ and $m.id2 \leftarrow id$
14:                 $history.add(m)$ and $outgoing.add(m)$
15:             else
16:                 for each $h \in history$ do             ▷ All possible bubble ends
17:                     if $h.id1 = m.id1$ then
18:                         $apply\_heuristic(m, h)$
19:                         if m is bubble then
20:                             mark $m.id2$ and $history.add(m)$
21:                         else
22:                             mark $h.id2$
23:     $send(outgoing)$                                  ▷ send message to all out-edges
```

along outgoing edges. This means that only in-edges are read, and the out-edges are written. As this algorithm implies an external BFS traversal, the I/O cost is $O(BPath(|V| + |E|)/B) \approx O(BPath|E|/B)$ where $BPath$ is the longest length among all bubbles and $|E| = O(|V|)$ because a dBG is a sparse graph. Then the total I/O cost is $O(limit * BPath * |E|/B) = O(BPath * |E|/B)$. On the other hand, the backward stage uses another BFS but in the opposite direction to the forward phase, so the I/O cost is the same. Additionally, it iterates through the set of vertices (lines 6–8) and executes a contraction on the resulting graph. Therefore, the I/O cost of the backward stage is $O((|E|/B + sort(|E|)) \log BPath)$.

At this point, the graph should be formed by contracted vertices. As each vertex represents a contig, we can output them using a full scan over the graph. Thus, the I/O complexity is $O(|V|/B)$, where $|V|$ is the number of contigs.

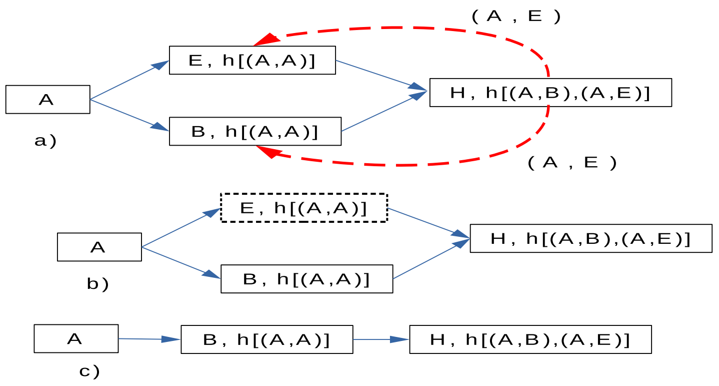

Fig. 4. Backward Bubble Elimination. a) H is a bubble ending, so it sends messages to the incoming neighbors communicating that E is marked. b) When E receives the message, it is marked and does not send more messages because it detects that the next node is the bubble. B does not send any message because its history does not have the node E. c) The graph is simplified after all bubbles are removed.

Algorithm 5. Backward step bubble detection

Input:
$G = (V, E)$:compressed dBG and *limit*: max. length of the bubble path
Output: G': a graph with all bubbles nodes marked.

```
1: procedure BackTrackPath($G$)
2:     $superstep \leftarrow 0$
3:     while $superstep \leq limit$ do
4:         $RemovePath(G, u)$
5:         $superstep \leftarrow iter + 1$
6:     for each $u \in V$ do                          ▷ External Memory Context
7:         if $u.mark = true$ then
8:             $delete(v)$
9:     $CompressGraph(G))$
10: procedure RemovePath($G$,$u$)                     ▷ External Memory Context
11:     if $superstep = 0 \wedge d^-(u) \geq 2$ then    ▷ Identify all possible bubble ends
12:         for each $m \in history$ do
13:             if $m$ is marked then
14:                 $outgoing.add(m.id, m.id2)$
15:     else
16:         $outgoing \leftarrow \{\}$; $incomming \leftarrow receive(id1, id2)$
17:         for each $m \in incomming$ do         ▷ Find the closest common ancestor
18:             if $m.id2 = id$ then
19:                 $dst \leftarrow history.pop(m.id)$
20:                 if $dst \neq id$ then
21:                     $outgoing.add(m)$ and $mark = true$
22:     $send(outgoing)$
```

5 Conclusions

In this paper, we have proposed out-of-core algorithms for dealing with contig generation, one of the most critical steps of fragment assembly methods based on *de Bruijn* graphs. Besides presenting these algorithms, we have made an I/O analytical evaluation that shows that it becomes feasible to generate *de Bruijn* graphs to obtain the needed contigs, independently of the available memory.

The I/O cost studies show that graph simplification is one of the most expensive steps. Actually, we could expect it because this phase involves a more significant number of vertices and edges. To deal with that, one could choose to do the assembly without simplifying the graph. In this condition, a tip is a branch with low coverage and not just a vertex. Hence, it will be necessary to apply a list ranking from the dead-end to branching nodes.

Among other issues, we may cite that it is hard to estimate the number of executions related to each phase. Primarily it depends on the number of nodes in the graph, which itself depends on the properties of the read's datasets. As these values vary from one dataset and sequencing technology to another, the assembly algorithms execute each step, a fixed and empirical number of times. As future work, we may cite the evaluation of other graph simplification approaches targeting erroneous and non-recognizable structures, such as X-cuts.

Acknowledgments. This work is partially supported by grants from CNPq, CAPES and FAPERJ, Brazilian Public Funding Agencies.

References

1. Anderson, R.J., Miller, G.L.: A simple randomized parallel algorithm for list-ranking. Inf. Process. Lett. **33**(5), 269–273 (1990)
2. Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de Bruijn Graphs. In: Raphael, B., Tang, J. (eds.) WABI 2012. LNCS, vol. 7534, pp. 225–235. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33122-0_18
3. Chapman, J.A., et al.: Meraculous: de novo genome assembly with short paired-end reads. PLoS One **6**(8), e23501 (2011)
4. Chiang, Y.J., et al.: External-memory graph algorithms. Procs. ACM/SIAM Symp. Discr. Algorithm. (SODA) **95**, 139–149 (1995)
5. Chikhi, R., Limasset, A., Medvedev, P.: Compacting de Bruijn graphs from sequencing data quickly and in low memory. Bioinformatics **32**(12), i201–i208 (2016)
6. Chikhi, R., Rizk, G.: Space-efficient and exact de Bruijn graph representation based on a bloom filter. Algorithm. Mol. Biol. **8**(1), 22 (2013)
7. Jackman, S.D., et al.: Abyss 2.0: resource-efficient assembly of large genomes using a bloom filter. Genome Res. **27**(5), 768–777 (2017)
8. Kundeti, V.K., et al.: Efficient parallel and out of core algorithms for constructing large bi-directed de Bruijn graphs. BMC Bioinf. **11**(1), 560 (2010)
9. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: large-scale graph computation on just a PC. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 31–46 (2012)

10. Kyrola, A., Shun, J., Blelloch, G.: Beyond synchronous: new techniques for external-memory graph connectivity and minimum spanning forest. In: Gudmundsson, J., Katajainen, J. (eds.) Experimental Algorithms — SEA 2014. Lecture Notes in Computer Science, vol. 8504, pp. 123–137. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07959-2_11
11. Li, Y., Kamousi, P., Han, F., Yang, S., Yan, X., Suri, S.: Memory efficient minimum substring partitioning. Proc. VLDB Endow. **6**(3), 169–180 (2013)
12. Malewicz, G., et al.: Pregel: a system for large-scale graph processing. In: Process ACM SIGMOD Intl. Conf. on Manage. Data, pp. 135–146 (2010)
13. McCune, R.R., Weninger, T., Madey, G.: Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Comput. Surv. (CSUR) **48**(2), 25:1–25:39 (2015)
14. Medvedev, P., Georgiou, K., Myers, G., Brudno, M.: Computability of models for sequence assembly. In: Giancarlo, R., Hannenhalli, S. (eds.) Algorithms in Bioinformatics — WABI 2007. Lecture Notes in Computer Science, pp. 289–301. Springer, Cham (2007). https://doi.org/10.1007/978-3-540-74126-8_27
15. Meng, J., Seo, S., Balaji, P., Wei, Y., Wang, B., Feng, S.: Swap-assembler 2: optimization of de novo genome assembler at extreme scale. In: Proceedings of the 45th ICPP, pp. 195–204. IEEE (2016)
16. Miller, J.R., Koren, S., Sutton, G.: Assembly algorithms for next-generation sequencing data. Genomics **95**(6), 315–327 (2010)
17. Salikhov, K., Sacomoto, G., Kucherov, G.: Using cascading bloom filters to improve the memory usage for de Brujin graphs. Algorithm. Mol. Biol. **9**(1), 2 (2014)
18. Simpson, J.T., Pop, M.: The theory and practice of genome sequence assembly. Ann. Rev. Genomics Hum. Genet. **16**, 153–172 (2015)
19. Sohn, J., Nam, J.W.: The present and future of de novo whole-genome assembly. Briefings Bioinf. **19**(1), 23–40 (2016)
20. Stephens, Z.D., et al.: Big data: astronomical or genomical? PLoS Bio. **13**(7), e1002195 (2015)
21. Ye, C., Ma, Z.S., Cannon, C.H., Pop, M., Douglas, W.Y.: Exploiting sparseness in de novo genome assembly. BMC (BioMed Central) Bioinf. **13**, S1 (2012)
22. Zeh, N.: I/o-efficient graph algorithms. In: EEF Summer School on Massive Data Sets (2002)
23. Zerbino, D.R., Birney, E.: Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Gen. Res. 821–829 (2008)