Partitioning Very Large de Bruijn Graphs for Genome Assembly

Author: Julio O. Prieto Entenza[1]
Advisor: Sérgio Lifschitz[1]

[1]Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)

{jentenza, sergio}@inf.puc-rio.br

Level: Doctoral Degree
Admission: March 2015
Expected Conclusion: February 2019

Abstract. *The genome assembly is a fundamental problem in bioinformatics. For de novo assembly procedures, without a reference genome, the de Bruijn graph is used to perform a computational evaluation. The datasets produced by current instruments can reach billions of reads and a few terabytes of data. As a consequence, the graph processing involves large data management techniques, such as distribution and parallelism, to allow feasible and efficient solutions. However, the level of parallelism in the construction of the contigs is limited because the graph traversal methods need to visit many partitions to create them. Thus, the consequence is a rise in the volume of communication between the processing nodes. This doctoral research proposes a new approach to decrease the communication during graph traversals in large volumes of data, through a partitioning according to the de Bruijn graph properties and their relationships to contigs.*

Resumo. *A montagem de fragmentos de genoma é um problema fundamental na bioinformática. Na montagem de novo, onde não existe uma cadeia de referência, é usada a estrutura de dados do grafo de Bruijn para realizar o processamento computacional. Os dados produzidos pelos instrumentos podem atingir bilhões de reads e alguns terabytes de dados. Consequentemente, o processamento do grafo envolve técnicas de gestão de grandes volumes de dados, como distribuição e paralelismo, permitindo soluções viáveis e eficientes. No entanto, o nível do paralelismo na construção dos contigs, que fazem parte do resultado da montagem, é limitado pois os algoritmos para percorrer o grafo podem visitar vários nós. Muitas soluções computacionais foram desenvolvidas para diminuir a comunicação entre os nós, fundamentalmente mecanismos de bloqueio e particionamento baseado em funções de hashing. Entretanto, é observado um aumento do volume de comunicação entre os nós de processamento. Este trabalho de pesquisa de doutorado propõe uma nova abordagem para diminuir a comunicação durante os percursos de grafos envolvendo grandes volumes de dados, em um particionamento de acordo com as propriedades do grafo de Bruijn e seu relacionamentos com os contigs.*

1. Introduction

Genome sequencing is the process of determining the order of nucleotides within a DNA molecule. Nowadays, it is a fundamental pillar for biological research, medical diagnosis, and biotechnology. However, current technologies for DNA sequencing cannot read whole genomes in a single run. Thus, it breaks a genome into a set of many short sequences (*reads*), string over the alphabet $\Sigma = \{A, C, G, T\}$, whose whole sequence is then reconstructed. The genome assembly problem consists in to combine these reads to reconstruct the original DNA sequences without a reference genome[Nagarajan and Pop 2009].

In practice, assemblers output several *contigs* which are an accurate reconstruction of a region from a DNA sequence. Therefore, the problem to solve is also named *contig assembly problem* [Simpson and Pop 2015]. For the resolution of the *contig* assembly problem, the main approaches are based on the *de Bruijn graph* (dBG). In this type of assembly, we break each read into a sequence of overlapping *k*-length substrings (*k-mers*). Later, we add the distinct *k-mers* as an edge links graph's vertices and those *k-mers* whose positions are adjacent in a read. We may formulate the assembly problem as finding a graph traversal that visits each edge in the graph once (Eulerian tour) or at least one (Chinese Postman tour) [Medvedev et al. 2007].

There is a growing gap between the output of the new generation of massively parallel sequencing machines and the ability to analyze the resulting data. The datasets produced by current instruments can reach billions of reads and represent hundreds of gigabytes or even terabytes of data [Schmidt and Hildebrandt 2017]. Dealing with this tremendous amount of information requires either to use substantial computational resources or to conceive specific algorithms and data structures designed for resource efficiency. As sequencing cost continues to decrease, sequencing very large genomes become affordable, but an assembly of such genome is barely possible. Consequently, to face the challenge to produce correct assemblies, the future assemblers will have to handle larger and larger datasets, to deal with large genomes or meta-genomes while providing a high throughput to follow the sequencing rate.

In the *de Bruijn* case, there are two main computational challenges, the size of the graph and the massive parallelization to process it. The size of the graph is a bottleneck because the memory cost is proportional to the genome size and complexity. If bacteria genomes take only a few gigabytes of RAM large genomes, such as mammalian and plants, requires over tens to hundreds of gigabytes. In the case of the human case with approximately 3 Gbps of genome size, we could have 4.8 billion nodes and 384 billion arcs. As a comparison, the Facebook's social graph had roughly 1.39 billion nodes with over 1 trillion edges in 2015 [Ching et al. 2015]. It should be noted that there are organisms or collection of them, (such as plants and metagenomes) with a genome much larger than the human genome. For instance, a 1,000 Genomes dataset with 200 terabytes of data can generate about 2^{47} *k-mers* (or nodes), 64-128 times larger than the problem size of the top result in the Graph 500 list [Meng et al. 2016].

On the other hand, several contributions were proposed to offer fast genome assembly through the use of massive multi-core servers. Since most assembly algorithms consist mostly of graph traversal operations, it is difficult to increase the throughput by optimizing these operations since they mainly rely on memory accesses. One of the main issues is that de Bruijn graph is sparse (e.g., for humans the dBG would be a 3×10^{-9} x

3×10^{-9} adjacency matrix with 2-8 non-zeros per row) and an extremely high diameter graph. As a result, the genome assembly computation is dominated by irregular memory access patterns and fine-grained synchronization. This situation leads to very high memory usage and more generally very high resource consumption systems.

Different solutions have been proposed to address the genome assembly problem. They vary from specific applications to distributed systems, NoSQL databases, and cloud computing [Sohn and Nam 2016]. Although their apparent differences, all of them have in common the use of graph systems to represent and process a dBG. Because of in the next future, the size of datasets will increase dramatically; this situation will stress the graph systems. Consequently, there is a need for new approaches to process all of this massive amount of information in a scalable way.

This work is structured as follows: we formalize next the technical challenges and the scientific problem (Section 2). Section 3 covers a description of some related works. We describe our proposed approach in Section 4. Later, a summarized methodology and the current state of the research are exposed in Section 5. Finally, Section 6 lists the expected contributions of our research work.

2. Problem statement

The challenge to enable massive parallelization is to limit the processes communications. The question of memory access is also critical when the massive parallelization requires the use of multiple machines communicating via very slow network accesses. If there is a partitioning function that could accurately predict how *k-mers* are placed into *contigs*, we could create a partitioning scheme in such way that we would map the *k-mers* belonging to the same *contig* to the same node. Thus, during the traversal, a processor would not incur in communication costs since none of the *k-mers* (e.g., lookups in a distributed hash table) that build up a particular *contig* are local to a single processor (local buckets in the distributed hash table). This idea is similar to general graph partitioning, which aims to minimize the number of edges between separated components. However, the main difficulty is that we do not know which are the *k-mers* that belong to a *contig*.

Therefore, we may define our **scientific problem** as *Is it possible to decrease the communication cost in a very large de Bruijn graph during the genome assembly process?* If yes, we could also reduce the overall cost of the *contig* generation process.

3. Related works

Despite the several works about dBG for genome assembly, the partitioning of the graph to decrease the communication cost has received inadequate attention. The standard approach is to distribute the *k-mers* evenly among the processors through a hash function [Jackman et al. 2017][Meng et al. 2016] to keep a well-balanced partitioning. Although this method is efficient, scalable and creates balanced partitions, it presents poor locality concerning *contigs*. The main problem with the hash approach is that it could map adjacent *k-mers* to different partitions. A critical property of adjacent *k-mers*, ignored by the hash functions, is that they share a common substring called *minimizer*. Thus, when parallel processes need to iterate over the *k-mers* to create *contigs*, they incur in higher communication costs, memory usage, and synchronization costs because they need to communicate to other partition where an adjacent *k-mer* exist.

Other approaches based on a minimizer have been proposed to bypass the hash problem by exploiting the ability of the minimizer to use sequence contiguity while binning the *k-mers*. The Minimum Substring Partition [Li et al. 2013] breaks the short reads into multiple small partitions based on a minimum p-substring of the *k-mers*. This partitioning allows consecutive *k-mers* to be distributed in the same partition and decreasing the number of I/O operations. Recently, BCALM2 [Chikhi et al. 2016] distribute the *k-mers* over disk partitions but with the direct construction of a compact dBG by the compaction of consecutive *k-mers* that constitute a simple path in the graph.

Although these approaches increase the locality through the minimizers for processing the dBG, their main drawback is that the same *k-mer* could be distributed (copied) on different partitions. The minimizer of a *k-mer* depends on its neighborhood which, in turn, depends on the read where the *k-mer* exists. Thus, when a processor selects a *k-mer*, it needs to communicate with different partitions to traverse the dBG. If we take into account that the number of unique *k-mers* can be in the orders of billions, then the number of cross-partition communication could be high.

4. Proposed approach

Before presenting our solution, we need to present some basic definitions and terminology involving graphs, particulary the *de Bruijn graph*.

A short read is a string over alphabet $\Sigma = \{A, C, G, T\}$. A *k-mer* is a string whose length is k. Let $G = (V, E)$ be a de Bruijn graph (dBG) with a set of vertices V and a set of edges E. Each vertex $u \in V$ represents exactly a *k-mer* and each edge $(u, v) \in E$ is a binary relation between two vertices such as $u \rightarrow v$ exists iff $suf_k(u) = pref_k(v)$. Let a *unitig* be a path $x_1, \ldots, x_m$ in the graph where all out- and in-degree of the nodes x_i for all $1 < i < m$ are equal to 1, and the in-degree of x_m and the out-degree of x_1 are 1. A *unitig* is said to be maximal if a vertex on either side can not extend it.

The l-minimizer of a string u is the lexicographical smallest l-mer that is substring of u [Li et al. 2013]. We define $lmin(u)$ (respectively $rmin(u)$) to be the l-minimizer of the $(k-1)$-prefix (respectively suffix) of u. If $u \rightarrow v$ then $rmin(u) = lmin(v)$ [Chikhi et al. 2016].

Two strings u and v are compactable in a set K if $u \rightarrow v$ and u is the only in-neighbor of v , and v is the only out-neighbor of u.

Methodology

The intuition behind our solution is based on a graph partitioning where each *k-mer* that belong to the same *contig* should be in the same partition. Despite the vast works done in graph partitioning [Buluç et al. 2016], in our case, the main issue is to find out how to determine if two k-mers belong to the same *contig*. However, we do not know the *contigs* a priori. Consequently, we cannot also know which set of *k-mers* belongs to a *contig*. We must remember that finding the set of *contigs* is the goal of an assembly algorithm.

To obtain a solution to our problem, let us try to draw on the intuition from an example. Considering Figure 1, we can see that any edge-covering graph traversal should visit C (or any other of the labeled traversals, for that matter). After all, once a traversal enters the starting point of C, it has no other choice but to continue all the way through

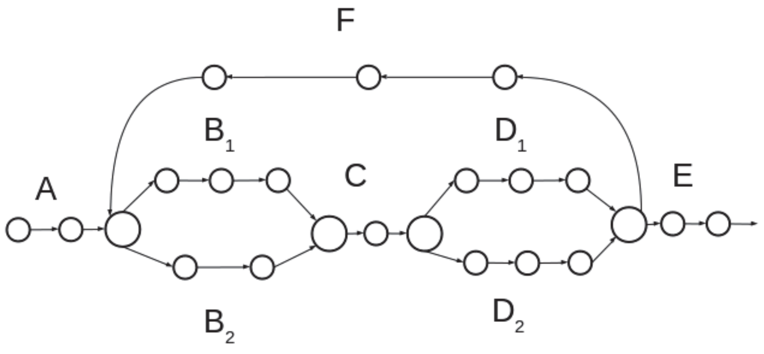

Figura 1. An example of a de Bruijn graph

the end of C. Remember that a *unitig* is a path such that all vertices except the first one have one incoming edge, and all vertices except the last one have one outgoing edge. As a maximal *unitig* is one that cannot be extended in either direction, the maximal *unitig* of a graph is a partition of the vertex set. Based on this intuition, we can define the algorithm, which outputs the partitions based on all the maximal *unitigs* in the graph.

A straightforward solution could use the BCALM2 method [Chikhi et al. 2016] to create the *unitigs* and, later, to make a partitioning over this set. However, if we apply this solution directly, we can lose the topological relationships among the maximal *unitigs* because these approaches ignore the nodes with more than two neighbors.

Consequently, our technique extends the BCALM2 approach and consists of three phases: 1) clustering according to the minimizers 2) contraction of the graph according to the *unitigs* and 3) cluster in the merge phase.

Algorithm 1 General algorithm

```
1: for all $x \in K$ do in parallel
2:     Write $x$ to $F(lmm(x))$
3:     if $lmm(x) \neq rmm(x)$ then
4:         Write $x$ to $F(rmm(x))$
5:     end if
6: end for
7: for all $i \in \{1, \dots, 4^l\}$ do in parallel
8:     Run $ContractBucket(i)$
9: end for
10: {$\delta$ user defined threshold}
11: $MergeComponents(\delta)$
```

Algorithm 2 $ContractBucket(i)$

```
1: Load $F(i)$ into memory.
2: $U \leftarrow i$-compaction of $F(i)$
3: $\forall u \in U$ initialize a forest $T = \{u\}$
4: repeat
5:     Find $t_i$ and $t_j$ who have a common minimizer
6:     $t_k \leftarrow t_i \cup t_j$
7:     delete $t_i$ and $t_j$
8:     $T \leftarrow T \cup \{t_k\}$
9: until there is not elements to reduce
```

In the first stage, the algorithm distributes the k-mers (K) to files $F(1), \dots, F(4^l)$. These are called bucket files. Each k-mer $x \in K$ goes into file $F(lmm(x))$, and if $lmm(x) \neq rmm(x)$, also in $F(rmm(x))$ to avoid false *unitigs* creation. The parameter l controls the minimizer size. This process is the same with BCALM2 [Chikhi et al. 2016].

In the second stage of the algorithm, we process each bucket file using the *ContractBucket* procedure. After the *k-mer* distribution of the first stage, the bucket file $F(i)$ contains all the k-mers whose left or right minimizer is i. Therefore, we can load $F(i)$

into memory and perform i-compaction and reduction on it. Then, we need to unite the different components within each partition because they represent partial *contigs*. We find all the branch nodes and add all the compacted edges to the forest. We repeat a similar process of finding the incident edge from each tree constructed so far to a different tree, and adding all of those edges to the forest. When it does, the set of edges forms a spanning forest. Since the size of the bucket is small, this compaction and reduction can be performed using in-memory algorithms. The resulting strings are then written to disk and will be processed during the third stage. At the end of the second stage, when all *ContractBucket* procedures are finished, we have performed all the necessary compressions and reductions on the data.

Algorithm 3 $MergeComponents(\delta)$

```
1: $\forall T$ initialize cluster $c = \{T\}$
2: repeat
3:     Find $c_i$ and $c_j$ who share the same k-mer
4:     $c_k \leftarrow c_i \cup c_j$
5:     delete $c_i$ and $c_j$
6:     $C \leftarrow C \cup \{c_k\}$
7: until $|C| < \delta$
```

At this stage of the algorithm, the *k-mers* $x \in K$ with $lmm(x) \neq rmm(x)$ exist in two copies. These *k-mers* are always at the ends (prefix or suffix) of the compacted strings, never internal, and they can be recognized by the fact that the minimizer at that end does not correspond to the bucket where it resides [Chikhi et al. 2016]. As we have generated more components (forest) than the number of computers in the 2nd phase, we choose an adjacent cluster and pack it into the same container during the merge algorithm, We keep a small number of cross-cluster edges and double *k-mers*.

This method resembles a hierarchical agglomerative clustering [Clauset et al. 2004] because we build the hierarchy bottom up. Hence, we find two adjacent clusters and merge them if the number of double *k-mers* decrease and the number of *unitigs*, within the new cluster, increase. Then, we repeatedly merge two new pairs of clusters until there is no adjacent cluster, or we reach the number of partitions.

5. Methods

The thesis subject arose from a demand for assembling sugarcane sequences as a part of research cooperation between PUC-Rio's BioBD Laboratory and the Laboratory of Molecular Biology of Plants (IBqM), Institute of Medical Biochemistry at UFRJ. One goal is to study the sugarcane genome into Brazilian species, which is very complicated as there are high rates of heterozygosity and repetitions, demanding a high availability of main memory.

An initial bibliographical survey has been carried out looking for ad-hoc data structures, partitioning and clustering approaches that impact parallel dBG Traversal algorithms. Also, three assembling tools Abyss [Jackman et al. 2017], MSP [Li et al. 2013] and BCALM2 [Chikhi et al. 2016] were thoroughly studied, to understand how the k-mers can be allocated to preserve the *contig* relationship.

All of these activities allows one to have more in-depth knowledge and understanding of the problem, defining the variables involved besides some bounds and limits.

6. Contributions

This thesis proposal will bring a set of additional contributions as listed below:

- Identification and formalization of the main variables that impact the parallel graph traversal algorithms on dBG assemblers.
- A new approach of dBG partitioning based on minimizers.
- A survey of dBG assemblers, emphasizing the approaches on parallel and distributed traversal algorithms used.
- An actual implementation of the proposed approach.

Referências

Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., and Schulz, C. (2016). Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer.

Chikhi, R., Limasset, A., and Medvedev, P. (2016). Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*.

Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., and Muthukrishnan, S. (2015). One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815.

Clauset, A., Newman, M. E., and Moore, C. (2004). Finding community structure in very large networks. *Physical review E*, 70(6):066111.

Jackman, S. D., Vandervalk, B. P., Mohamadi, H., Chu, J., Yeo, S., Hammond, S. A., Jahesh, G., Khan, H., Coombe, L., Warren, R. L., and Birol, I. (2017). ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter. *Genome Research*.

Li, Y., Kamousi, P., Han, F., Yang, S., Yan, X., and Suri, S. (2013). Memory efficient minimum substring partitioning. In *Proceedings of the VLDB Endowment*, volume 6, pages 169–180. VLDB Endowment.

Medvedev, P., Georgiou, K., Myers, G., and Brudno, M. (2007). Computability of models for sequence assembly. In *International Workshop on Algorithms in Bioinformatics*, pages 289–301. Springer.

Meng, J., Seo, S., Balaji, P., Wei, Y., Wang, B., and Feng, S. (2016). Swap-assembler 2: Optimization of de novo genome assembler at extreme scale. In *Parallel Processing (ICPP), 2016 45th International Conference on*, pages 195–204. IEEE.

Nagarajan, N. and Pop, M. (2009). Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *J Comput Biol.*

Schmidt, B. and Hildebrandt, A. (2017). Next-generation sequencing: big data meets high performance computing. *Drug Discovery Today*, 22(4):712 – 717.

Simpson, J. T. and Pop, M. (2015). The theory and practice of genome sequence assembly. *Annual review of genomics and human genetics*, 16:153–172.

Sohn, J.-i. and Nam, J.-W. (2016). The present and future of de novo whole-genome assembly. *Briefings in bioinformatics*, 19(1):23–40.