# Performance Assessment by Stages of Main Genomic De-Novo Assemblers Based Upon De Bruijn Graphs

Nelson Enrique Vera-Parra[1], José Nelson Pérez-Castillo[2], Cristian Alejandro Rojas-Quintero[3]

[1] GICOGE Research Group - Teacher / Researcher
Distrital University Francisco José de Caldas, Bogotá D.C., Colombia, Zip Code: 111321, Tel: +5713239300
[2] GICOGE Research Group - Director Center for Scientific Research and Development
Distrital University Francisco José de Caldas, Bogotá D.C., Colombia, Zip Code: 111321, Tel: +5713239300
[3] GICOGE Research Group - Student
Distrital University Francisco José de Caldas, Bogotá D.C., Colombia, Zip Code: 111321, Tel: +5713239300
neverap@udistrital.edu.co

**Abstract:** This article documents the performance assessment of main genomic assemblers based upon De Bruijn Graphs and their respective stages. The objetive is to identify and compare computational requirements, advantages and bottlenecks on each of the assembly algorithms steps for every tool evaluated with the purpose to provide a reference frame and detect future computational challenges to improve these techniques.

The assessed assemblers were: Abyss, Velvet, SOAPdenovo2, Minia and EPGA2. The dataset used to perform the assessment was: 64587949 reads of 101bp corresponding to the 14th human chromosome. The measured variables were: read and write operations, use of main memory (RAM), parallelization (number and percentage of used cores) and processing time. All measures were taken with two kmer sizes: k=31 and k=55. The results show: the assemblers that use partitioning techniques for kmers counting reduces considerably the memory use, but increase the amount of I/O operations; the use of techniques for graph simplification by parts allows to reduce substantially the memory requirement in the contigs generation step, however they increase the processing time, the use of error removal reduce the amount of necessary memory in future stages; the data structures used to represent the graph directly affect the RAM use, as an example, Minia reduces substantially the used RAM peak through the use of bloom filters in cascade.

[Nelson Enrique Vera-Parra, José Nelson Pérez-Castillo, Cristian Alejandro Rojas-Quintero. **Performance Assessment by Stages of Main Genomic De-Novo Assemblers Based Upon De Bruijn Graphs.** *Life Sci J* 2015;12(12):13-21]. (ISSN:1097-8135). http://www.lifesciencesite.com. 3. doi:10.7537/marslsj121215.03.

**Keywords:** Bioinformatics, De Bruijn Graphs, Genome assembly, Next Generation Sequencing

## 1. Introduction

The rapid fall of sequencing costs since the arrival of the next generation sequencing (NGS) allowed the possibility to sequence complete genomes of different organisms with the aim to ease the study of their genes. One of the main problems and challenges is in the de-novo assembly stage(Millller, Koren & Sutten, 2010), (Vera-Parra, Perez-Castillo & Rojas-Quintero, 2015), in this stage short reads are used ( between 100bp and 500bp) from machines with Illumina or Roche 454 technology and are assembled in contigs and metacontigs in such a way that as a result a complete assembled genome is obtained without need to count with a reference genome. For the assembly of small genomes, such as the bacterial genome assembly, these tools has good performance and most of the time require a little computation time. However as the genome size increases (eg the chromosome of a mammal) the computing requirements increases to perform this process (Zhang et al., 2011).

Computational intensity required to perform an assembly without a reference genome ( de-novo) has been addressed in the last decade using techniques based on De Bruijn graphs (Compeau, Pevzner & Tesler, 2011). Actually, the most used genomic assemblers are based on this approach, such as Velvet (Zerbino & Birney, 2008), ALLPATHS (Butler et al., 2008), ABySS (Simpson et al, 2009), SOAPdenovo2 (Luo et al, 2012), MINIA (Chikhi & Rizk 2013).

The assemblers that use De Bruijn graphs generally have similar workflows. Below the tipical stages of these workflows are described:

*Kmer generation and counting*: During this stage, the reads are divided in shorter fragments of k size, called kmers, to perform a counting of ocurrences of each of these. This is one of the stages with the more RAM demand because it is necessary to store these kmer in main memory (Zhang et al, 2014).

*De Bruijn graph construction*: From the found kmers a De Bruijn graph is built which later will be used to determine the contigs.

*Sequence determination using the De Bruijn graph*: Once De Bruijn graph is built and stored in main memory a simplification of itself done with the purpose to find contigs and metacontigs. Some

assemblers do a first simplication of the graph to generate the contigs and then make use of the paired-end information to link these contigs in metacontigs with more size.

In previous works such as Assemblathon 1 (Earl D., 2011), Assemblathon 2 (Bradnam et al, 2013) , and Genome Assembly Gold-Standard Evaluations (GAGE) (Salzberg et al, 2012) performance tests were conducted to the assemblers regarding to their results, that is, the number of generated contigs and their statistics. However in these works few mentions or none about the variables such as read/write operations, the RAM use and the parallelization of these assemblers. Moreover, some of the performance assessments available focus in the general performance analysis without focusing on each of its stages.

In this article approach to assess several genomic assemblers based on De Bruijn graphs, they were selected on two criteria: high use and variety of techniques used to reduce the computational requirement of building, representing and graph processing. For each assembly stage data related to read and write operation, main memory use (RAM), parallelization (number and percentage of used cores) and processing time are measured with the purpose to verify which tools has greater computational requirements.

## 2. Material and Methods

### 2.1 Tools to assess

The tools to assess are the assemblers based on De Bruijn Graphs that only use short paired-end reads to determine contigs. Assemblers were chosen in such a way that diversity would be present on the k-mers counting algorithms in the structures to represent the graph in memory and these had greater use during the last decade. Note that assemblers such as ALL-PATHS-LG were not taken in count in this assesment because this one requires long fragments as an additional to short fragments to be executed.

*ABySS*: Is a de novo parallel assembler for paired sequences, specifically designed for short reads. Roughly the algorithm used by this tool has the following stages: First, without using the paired-end information, contigs are extended until either they cannot be unambiguously extended or come to a blunt end due to a lack of coverage. In the second step the paired-end information is used to resolve ambiguities and merge contigs (Simpson et al, 2009). ABySS is implemented in C++ and use the openmpi library (in some of its stages) with the purpose to allow the comunication bewteen several computer nodes and ease its execution on a cluster. Additionally AbySS use the google-sparsehash library to decrease the RAM requirement in the kmers counting.

*Velvet*: Is a de novo assembler for short reads. The Velvet's algorithm mainly has two stages: Hash production (hashing) and graph construction; these stages are done by velveth and velvetg respectively. Velveth reads the sequences files and builds a dictionary with all the words of size k, along with the local aligments definition between these lectures. Subsequently, velvetg read these aligments, builds the De Bruijn graph, remove erros, simplify the graph and resolve the repetition based on the parameters supplied by the users. Velvet is implemented in C++ and use the OpenMP library with the purpose that some assembly stages can be run in paralallel in the same computational node.

*SOAPdenovo2*: Is a novel short-read assembly method that can build a de novo draft assembly for the human-sized genomes. This algorithm has the following steps: De Bruijn Graph (DBG) construction, contig assembly, paired-end (PE) reads mapping, scaffold construction, and gap closure. SOAPdenovo2 is implemented in C++ and for some of its processing stages use several threads.

*Minia*: Is a short-read assembler based on a De Bruijn graph, capable of assembling a human genome on a desktop computer in a day. The output of Minia is a set of contigs. The main stages for Minia's algorithm are: kmers counting, graph construction route and simplification of it. For the kmers counting stage Minia use bloom filters to reduce the RAM memory. Minia is implemented in C++. Note that unlike Velvet and AbySS, Minia does not use the available paired-end information to build metacontigs.

*EPGA2*: Is a genomic assembler oriented to efficient memory use. To achieve this purpose uses BLESS (Heo et al, 2015) to fix reads errors, DSK (Rizk, Lavenier & Chikhi, 2013) for kmers counting and BCALM (Chikhi et al, 2014) to simplify De Bruijn graph nodes. Once the nodes are simplified and contigs generated EPGA2 merge the contigs in a parallel way (Luo et al, 2015).

### 2.2 Datasets

The dataset used to this assessment corresponds to 64587949 short reads (101bp), paired-end of the 14th Homo Sapiens chromosome.

### 2.3 Computer Equipment

The computer where this assessment was in place has the following described characteristics: Operating System: Debian Wheezy (AMD64). Processor: Intel(R) Xeon(R) cpu E7450 @ 2,40GHz, 24 cores. RAM: 64GB. Hard Disk: 160GB HDD.

## 3. Results

### 3.1 Results by tool

**Velvet performance by stages**

Below is described each of the Velvet's execution stages, highlighting the relevant aspects regarding to its computational requirements for k=31 (Figure, 1) and k=55 (Figure, 2).

*Kmer load*: Velveth read the files with reads and generates a hash table of them. This stage has the greatest I/O use (with a peak of 8000 Kb/s), the RAM use during this stage is inversely proportional to the kmer size used.

*Pregraph, bubbles popping and splurs remotion*: The hash table previously created by velveth is loaded, splurs and bubbles are removed because of possible sequencing errors. This stage is the one with less execution time for k=31, k=55.

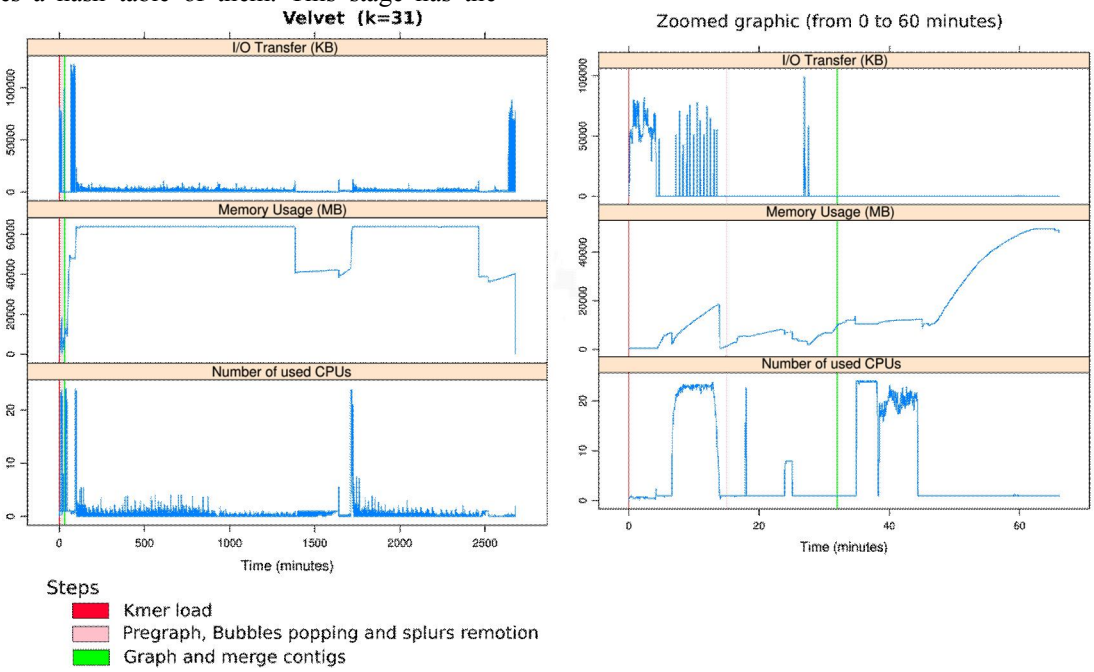


**Figure 1.** Velvet execution and its stages (k=31), in the right side zoom from minute 1 to 60.

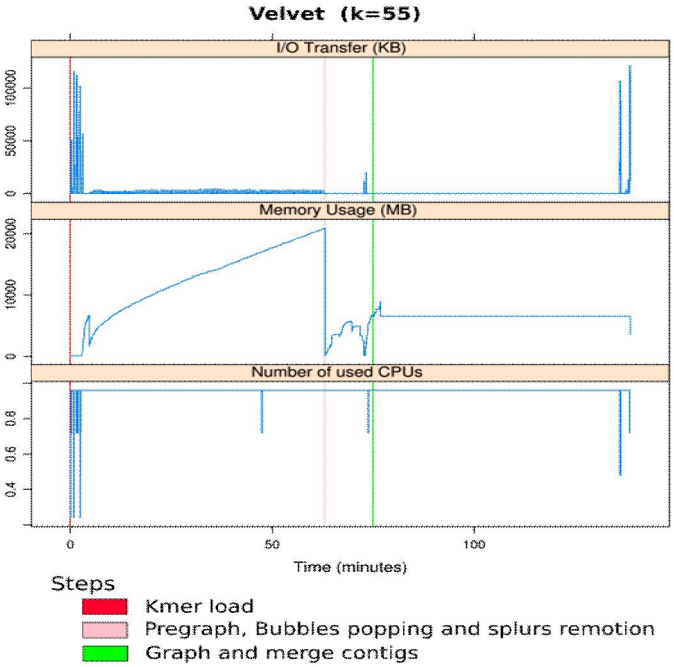


**Figure 2.** Velvet execution and its stages (k=55).

*Graph and contig merging*: The final graph is processed to generate the contigs. This stage was the one with the more execution time, it is remarkable in the case when k=31 because it needed to use the swap memory ( in addition to the 64 GB of the available RAM it required 10 GB of swap). Some sections in this stage are executed in parallalel.

**ABySS performance by Stages.**

The Figure 3 shows the execution of ABySS for k=31 and k=55. Below are described each of the AbySS execution stages, highlighting relevant aspects regarding its computational requirements.

*Kmer load*: Corresponds to the reads load and kmers extraction. It is the stage with major I/O transfer intensity (peak 10000KB/S) and major RAM requirement (for k=33 the peak is 7000MB, for k=55 the peak is 8000MB). This stage has a low parallelization.

*Bubbles popping and splurs remotion*: Once kmers are detected in this stage, the bubbles and splurs are removed from the graph with the purpose to reduce the RAM requirement in the next stages and obtain major reliability due to the error removal. This is the stage that require less execution time and does not have parallelization.

*Graph and merge contigs*: Graph construction is done and the deduction of contigs and metacontigs. This stage use less memory than the previous stages because of the use of the library sparsehash and show major parallelization.

**Minia performance by stages.**

The Figure 4 shows the execution of Minia for k=31 and k=55. Below is described each of the Minia's execution stages, highlighting the relevant aspects regarding to its computational requirements.

*Kmer load*: This stage uses DSK (Rizk, Lavenier & Chikhi, 2015) which is the kmer counter program oriented to low RAM use using the reads partitioning on disk technique. This stage is more intensive in terms of I/O with an average of 5000KB/S and moderate RAM (a peak of 4000MB for k=31 and a peak of 4200MB for k=55). Additionally, it is noted that this stage is highly parallelized.

*Bloom filter*: The purpose of the use of this filter is to remove kmers with lower coverage. It is observed that this stage has an average paralellization and it is the one with less execution time requirement for both k values.

*Graph and assembly*: Graph construction and contigs assembly. It is observed that this stage is the one with more time requirement and it has less parallelization. To represent the graph in memory bloom filters in cascade are used (Salikhov, Sacomoto & Kucherov, 2013). It is to note that Minia does not use the paired-end information and for this reason has less exection time in this stage in contrast with the others assemblers.

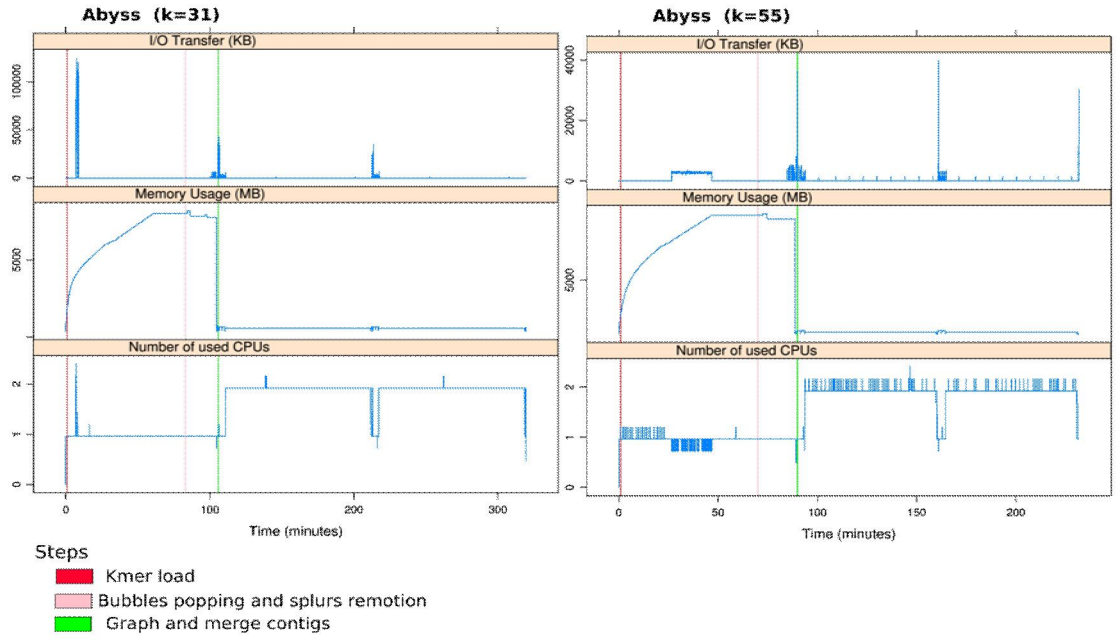


**Figure 3.** ABySS execution and its stages (k=31 and k=55)

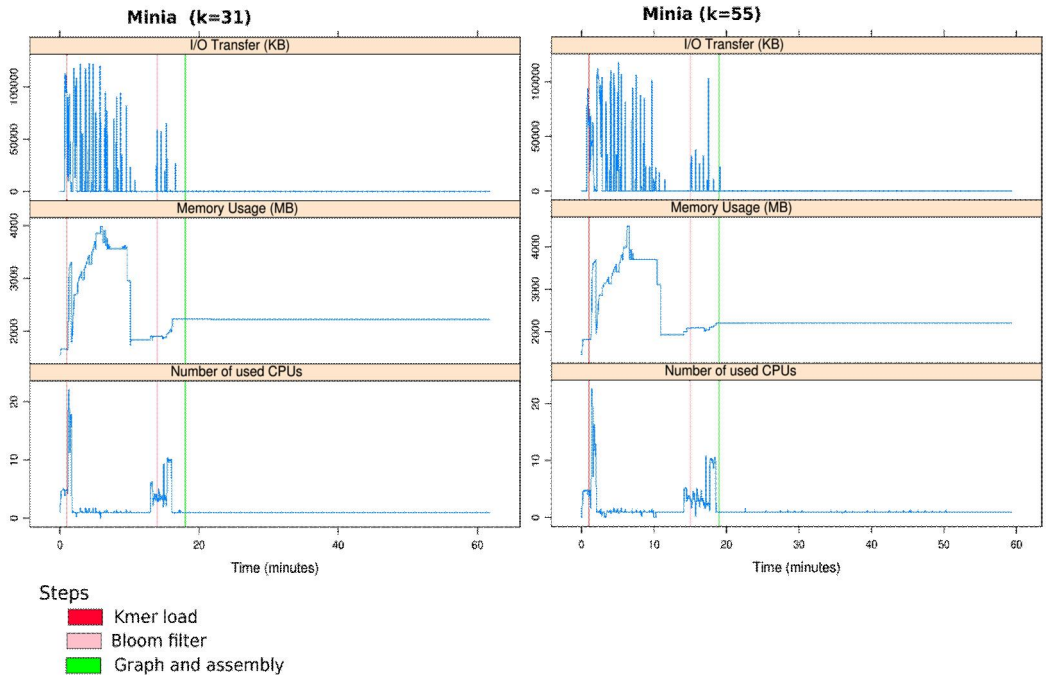


**Figure 4.** Minia execution and its stages (k=31 and k=55)

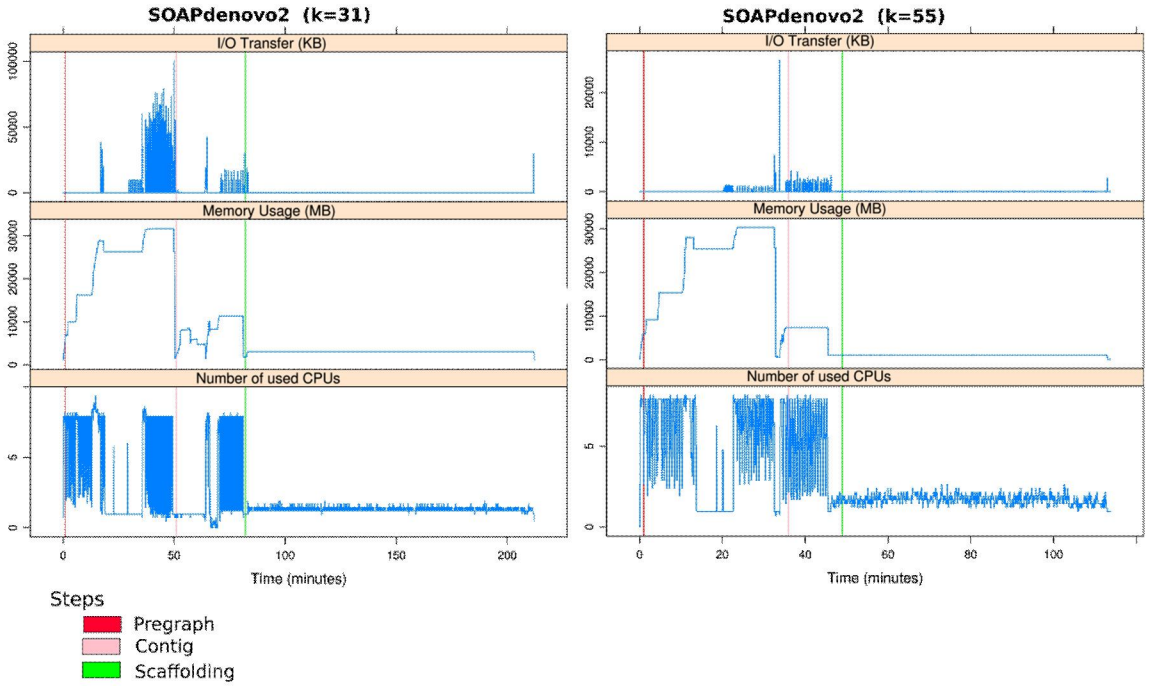


**Figure 5.** SOAPdenovo2 execution and its stages (k=31 y k=55)

**SOAPdenovo2 performance by steps**

The Figure 5 shows the execution of SOAPdenovo2 for k=31 and k=55. Below is described each of the SOAPdenovo2's execution stages, highlighting the relevant aspects regarding to its computational requirements.

*Pregraph*: Kmers are loaded in memory, possible errors are removed and finally counter kmers are dumped to disk. This stage has the greater I/O (with a peak of 10000KB/S for k=33 and 2000KB/S for k=55) and the greater RAM (with a peak of

30GB) usage for both k values and a high parallelization.

*Contig generation*: Contigs are generated from the preprocessed graph. This stage has high paralellization and moderate RAM usage (5GB in average).

*Scaffolding*: Generated contigs are used to superimpose them and obtain larger contigs. This is the stage less parallelized in the program execution and it has a moderate RAM usage (3GB in average).

**EPGA2 performance by stages.**

The Figure 6 shows the execution of EPGA2 for k=31, the execution for this tool was not possible for k=55 because it only supports a k size less than 32. Below is described each of the EPGA2's execution stages, highlighting the relevant aspects regarding to its computational requirements.

*Bless*: during this stage, a Bloom filter is applied with the purpose of fixing the reads. The execution of this stage is done sequentially, it does not have a pronounced usage of RAM and greater intensity in I/O occurs when loading and write the readings.

*DSK*: The same as Minia's case, DSK has an intensive I/O usage because it does disk partitioning for the kmers counting, its execution is sequential.

*Bcalm*: Bcalm is a tool for De Bruijn graph compacting focused to low memory use. Bcalm partition all kmers based on frequency of their minimizers. Finally uses each kmer as a node and compacts the simple paths of the graph. At the beginning of its execution Bcalm has high I/O intensity due to the kmer distribution on several files, it's memory peak usage is 1.5GB of RAM and its execution is sequential.

*EPGA*: During this stage all simple paths generated by Bcalm are integrated (contigs). This stage has low I/O intensity, a peak of RAM memory usage of 15 GB and it is totally paralellized.

### 3.2 Comparison between tools.

In the Table 1 average and peak of the computational requirements of the execution of each of the tools and their stages are compared.

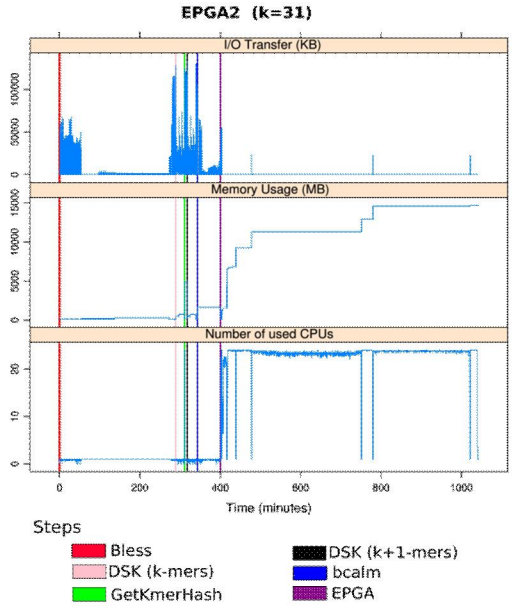


**Figure 6.** EPGA2 execution and its stages for k=31.

Table 1. Average and peak values for each stage and variable (I/O, RAM, Number of CPUs)

| Tool | Stage | RAM (MB) | | Number of CPUs | | | | I/O (KB/S) | | | | Time (Mins) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | k=31 | k=55 | k=31 | | k=55 | | k=31 | | k=55 | | k=31 | k=55 |
| | | Peak | Peak | Avg | Peak | Avg | Peak | Avg | Peak | Avg | Peak | | |
| Velvet | Kmer load | 18458 | 20893 | 11 | 24 | 1 | 1 | 7000 | 25000 | 600 | 37000 | 15 | 63 |
| | Pregraph, bubbles popping and splurs remotion | 9740 | 6781 | 2 | 22 | 1 | 1 | 250 | 30000 | 300 | 41000 | 17 | 12 |
| | Graph and contig merging | 63916 | 6781 | 2 | 24 | 1 | 1 | 400 | 40000 | 100 | 7000 | 2618 | 70 |
| Abyss | Kmer load | 8000 | 9230 | 1 | 3 | 1 | 1 | 450 | 40000 | 300 | 1300 | 83 | 70 |
| | Bubbles popping and splurs remotion | 8170 | 9315 | 1 | 1 | 1 | 1 | 80 | 12000 | 200 | 13500 | 23 | 20 |
| | Graph and merge contigs | 574 | 1733 | 1 | 2 | 2 | 2 | 90 | 13000 | 60 | 14000 | 224 | 155 |
| Minia | Kmer load | 3984 | 4492 | 2 | 20 | 2 | 23 | 5500 | 40000 | 5500 | 39000 | 15 | 15 |
| | Bloom filter | 2231 | 2201 | 3 | 11 | 4 | 10 | 700 | 22000 | 2300 | 37000 | 4 | 4 |
| | Graph and assembly | 2231 | 2206 | 1 | 1 | 1 | 2 | 270 | 33 | 60 | 8000 | 43 | 40 |
| SOAP Denovo2 | Pregraph | 31739 | 30274 | 4 | 10 | 4 | 8 | 1300 | 33000 | 100 | 12500 | 51 | 36 |
| | Contig generation | 11324 | 7328 | 2 | 9 | 4 | 8 | 300 | 13000 | 200 | 1400 | 31 | 13 |
| | Scaffolding | 3049 | 1102 | 1 | 2 | 2 | 3 | 40 | 10000 | 20 | 1250 | 133 | 65 |
| EPGA2 | Bless | 258 | N/A | 1 | 1 | N/A | N/A | 800 | 35000 | N/A | N/A | 289 | N/A |
| | DSK | 4924 | N/A | 1 | 1 | N/A | N/A | 7500 | 43000 | N/A | N/A | 54 | N/A |
| | bcalm | 1651 | N/A | 1 | 1 | N/A | N/A | 1200 | 25000 | N/A | N/A | 57 | N/A |
| | EPGA | 14671 | N/A | 23 | 24 | N/A | N/A | 40 | 20000 | N/A | N/A | 653 | N/A |

## 4. Discussions

### 4.1 Analysis by variable

**Analysis Based On RAM use.**

The stage that required major RAM usage in most assessed assemblers was the counting and kmers detection, however when this process use kmers partitioning in disk (technique used by Minia and EPGA2) it was observed a significant reduction in the memory usage. A previous processing of reads through bloom filters allows a reduction on memory usage because reads with possible sequencing errors are removed before graph generation (this was evident in the BLESS stage of EPGA2 and in the Bloom Filter stage of Minia).

One of the stages with less RAM usage was the contig and metacontig generation, however in Velvet's case this stage required more RAM because of the data structure used to represent the graph. The tool with less RAM usage in the contigs generation was EPGA2, due to the Bcalm tool use which use kmers partitioning on disk by minimizers frequency thus allowing graph simplification by parts.

The increase in the kmer length reduces the memory requirement in most stages of all the tools, except for the kmers load stage.

**Analysis Based On I/O**

The stage with greater I/O transfer observed in all assemblers was the kmers counting and detection stage because the reads should be loaded into memory and later the resulting kmers must be written to disk. It was further noted that the assemblers that use counting techniques based on partitioning on disk make more I/O transference due to a redistribution of kmers in intermediate files according to theirs minimizers.

The stage with the less average I/O transfer in most assemblers is the contigs and metacontigs generation stage, since during this stage the data structure that represents the graph is already loaded in main memory and it is being processed, hence the requirements for this stage are mainly of CPU. Unlike of the rest of the assemblers, EPGA2 presented a major demand of I/O transference for contigs production because it makes use of disk partitioning.

**Analysis Based On Parallelization**

Analyzing the average of used CPUs for each stage in each tool it was observed a medium/low degree of paralellization, except for the metacontigs generation step in EPGA2.

Analyzing the relation between the average and the peak of the used CPUs for each tool it became clear that for the vast majority of stages that presented medium or high paralellization values, these values only ocurred in moments that represent a low percentage of the total processing time of the stage.

Only the metacontigs generation stage in EPGA2 and the kmers load stage in Velvet (only for K=31) presented a high continuous parallelization. The kmers load stage and error removal on SOAPdenovo2 (for both K values) showed a high parallelization on the half of its processing time.

**Analysis Based On Execution Time**

The common stage in all the assemblers that required greater demand of processing time was the contigs and metacontigs generation because during this stage the simplification and traversal of the graph is done.

The stages with less execution time demand in the majority of the tools were the kmers load and error removal.

It was evident in all the tools that with a greater kmer size the less execution time is required. This is because there is a lower the amount of generated nodes in the graph.

The assembler that used the less execution time was Minia because it does not use the paired-end information to do the metacontigs generation.

### 4.2 General Conclusions

The kmer size directly affects in the amount of RAM and processing time to make an assembly: the more kmer size the less RAM requirement and less execution time.

The data structure used to represent the graphs directly affects the RAM memory usage; eg Minia has low demand due to the data structure used ( bloom filters in cascade) unlike ABySS that had a greater requirement due to the use of a Hash Table.

The use of techniques of error removal reduces the amount of RAM required. This can be seen in assemblers like Velvet and ABySS (Pregraph, bubbles popping and splurs remotion) or Minia and EPGA2 (through bloom filters).

The use of partitioning in disk techniques in the kmer counting stage reduces the amount of RAM required but increase the I/O transfer. This can be noted in assemblers as Minia and EPGA2 that use DSK for this stage.

The use of graph simplification by parts techniques allow to reduce substantially the memory requirement in the contigs generation stage, but increase the processing time. This can be seen in EPGA2 that use Bcalm.

### 4.3 Challenges

Below are listed the computational challenges that will allow improve the future genomic assemblers performance based on De Bruijn graphs. Some of these challenges were observed in the

assesed assembler shortcomings and some other by the strenght of the ones that are taking them in count:

Design or adapt data structures to reduce memory requirement and execution time in the graph processing and representation.

Design techniques for kmer partitioning on disk that generate homogeneous file sizes and group kmers in such a way that the analysis is favored.

Design graph simplification techniques that reduce RAM requirement and execution time.

Integrate the kmer counting techniques, simplification and graph construction such that a single partitioning process kmers disk is performed.

Design error removal techniques that avoid graph overlaps hence reducing processing time.

Condition the algorithms used in each stage of the assemblers so that the execution in parallel is enabled across heterogeneous platforms to make use of accelerators such as GPUs (Graphics Processing Units), DSPs (Digital Signal Processors), FPGAs (Field Programmable Gate Arrays), among others.

**Acknowledgements**

Work done in collaboration with High Performance Computational Center (CECAD) - Distrital University Francisco José de Caldas, Bogotá D.C., Colombia (http://cecad.udistrital.edu.co) and Genetics Institute - National University (IGUN), Colombia, (http://www.genetica.unal.edu.co).

**Corresponding Author:**
PhD(c) Nelson Vera-Parra
GICOGE Research Group - Teacher / Researcher
Distrital University Francisco José de Caldas, Bogotá D.C., Colombia, Zip Code: 111321
neverap@udistrital.edu.co

**References**

1. Bradnam, K. R., Fass, J. N., Alexandrov, A., Baranay, P., Bechner, M., Birol, I., ... & MacCallum, I. (2013). Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. GigaScience, 2(1), 1-31.
2. Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I. A., Belmonte, M. K., Lander, E. S., ... & Jaffe, D. B. (2008). ALLPATHS: de nsdovo assembly of whole-genome shotgun microreads. Genome research, 18(5), 810-820.
3. Chikhi, R., Limasset, A., Jackman, S., Simpson, J. T., & Medvedev, P. (2014, January). On the representation of de Bruijn graphs. In Research in Computational Molecular Biology (pp. 35-55). Springer International Publishing.
4. Chikhi, R., & Rizk, G. (2013). Space-efficient and exact de Bruijn graph representation based on a Bloom filter. Algorithms for Molecular Biology, 8(22), 1.
5. Earl, D., Bradnam, K., John, J. S., Darling, A., Lin, D., Fass, J., ... & Xie, Y. (2011). Assemblathon 1: a competitive assessment of de novo short read assembly methods. Genome research, 21(12), 2224-2241.
6. Heo, Y., Wu, X. L., Chen, D., Ma, J., & Hwu, W. M. (2014). BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads. Bioinformatics, btu030.
7. Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., ... & Wang, J. (2010). De novo assembly of human genomes with massively parallel short read sequencing. Genome research, 20(2), 265-272.
8. Li, Y., Kamousi, P., Han, F., Yang, S., Yan, X., & Suri, S. (2012). Memory efficient de Bruijn graph construction. arXiv preprint arXiv:1207.3532.
9. Li, Z., Chen, Y., Mu, D., Yuan, J., Shi, Y., Zhang, H., ... & Fan, W. (2012). Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. Briefings in functional genomics, 11(1), 25-37.
10. Luo, J., Wang, J., Li, W., Zhang, Z., Wu, F. X., Li, M., & Pan, Y. (2015). EPGA2: memory-efficient de novo assembler. Bioinformatics, btv487.
11. Luo, R., Liu, B., Xie, Y., Li, Z., Huang, W., Yuan, J., ... & Wang, J. (2012). SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. Gigascience, 1(1), 18.
12. Miller, J. R., Koren, S., & Sutton, G. (2010). Assembly algorithms for next-generation sequencing data. Genomics, 95(6), 315-327.
13. Pevzner, P. A., & Tang, H. (2001). Fragment assembly with double-barreled data. Bioinformatics, 17(suppl 1), S225-S233.
14. Pevzner, P. A., Tang, H., & Waterman, M. S. (2001). An Eulerian path approach to DNA fragment assembly. Proceedings of the National Academy of Sciences,98(17), 9748-9753.
15. Salikhov, K., Sacomoto, G., & Kucherov, G. (2013, February). Using cascading Bloom filters to improve the memory usage for de Brujin graphs. In WABI (pp. 364-376).
16. Salzberg, S. L., Phillippy, A. M., Zimin, A., Puiu, D., Magoc, T., Koren, S., ... & Yorke, J. A. (2012). GAGE: A critical evaluation of genome assemblies and assembly algorithms. Genome research, 22(3), 557-567.
17. Simpson, J. T., Wong, K., Jackman, S. D., Schein, J. E., Jones, S. J., & Birol, I. (2009). ABySS: a parallel assembler for short read

sequence data. Genome research, 19(6), 1117-1123.
18. Rizk, G., Lavenier, D., & Chikhi, R. (2013). DSK: k-mer counting with very low memory usage. Bioinformatics, btt020.
19. Vera-Parra N., Perez-Castillo J. & Rojas-Quintero C. (2015). Performance assessment for main stages in genomic and transcriptomic data processing based upon reads from illumina sequencing technologies. International Journal of Applied Engineering Research (IJAER), pp 34670-34674.
20. Zerbino, D. R., & Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome research, 18(5), 821-829.
21. Zhang, Q., Pell, J., Canino-Koning, R., Howe, A. C., & Brown, C. T. (2014). These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure.
22. Zhang, W., Chen, J., Yang, Y., Tang, Y., Shang, J., & Shen, B. (2011). A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. PloS one, 6(3), e17915.

11/26/2015