University Of Thessaly- Universidad Politecnica de Madrid
Faculty of Computer and Communication Engineering

Study of Short Read Genome Assemblers for Shared and Distributed Memory Systems

Vaios Noutsos
Supervisors:
Prof. Panagiota Tsompanopoulou
Prof. Vicente Martin

Madrid - July 2012

To my family and friends

Acknowledgments

First of all I would like to thank my supervisors because they gave me the opportunity to go on Erasmus, carry out my thesis, in the new for me field of Genomics, and also work in an environment like Madrid's supercomputing and visualization Center.

Moreover I would like to thank the people in CeSViMa because they were always very kind and helpful and without their assistance the completion of the current study would be almost impossible.

A special thanks goes to my co-worker Nikos for all the moments that we passed together in this experience and the support that we gave each other.

Last but not least, I would like to thank my parents, my sister, my grandparents and my friends Dimitris, Emmanouela, Maira, Lefteris, Stavros, Fokion, Christina because they always support me in these years in the good and the bad moments.

Contents

1 Introduction

1.1 What is a genome?

Genome specifies life. Genome is the sum of all the biological material that are necessary to make and maintain alive every organ, cell and tissue of an organism, from a human to a single celled organism. Almost every cell of a living organism has a copy of the genome. Some information provided form the genome is used in the same way from all the cells and some other consist in some characteristic of distinct types of cells.

There are two kinds of genome:

- *DNA genome*
  - Contained in unicellular and multicellular organisms

- *RNA genome*
  -Contained in some virus

The human genome, like all multicellular organisms consists in 2 components:

- The nuclear genome contained in every cell

- the mitochondrial genome contained in the mitochondrion of every cell

These two kinds of genomes are made from DNA molecules. There are thousands molecules in the nuclear genome and few molecules in the mitochondrial genome.

The DNA molecule consists in 4 different nucleotides. These nucleotides consist of a deoxyribose, a phosphate group attached to it, and a base. The base of a nucleotide can be one between: Adenine, Thymine, Guanine and Cytosine. Sequences of these nucleotides form every one of the two complementary DNA strands-helixes. Adenine's complementary base is Thymine and Guanine's complementary base is Cytosine. Complementary means that if a strand of the DNA has a nucleotide with, for example, an Adenine base the nucleotide from the opposite strand in the same position will have as a base a Thymine that will bond with Adenine. The way that is used to represent them in a graphic or text application is through their abbreviations A, T, G, and C.

The nucleotide sequence of the genome is present on each of the two strands of the double helix of DNA. Given one strand it's possible to construct the other due to complementarity. Each of the strands contains the same information.

Organization in a cell:

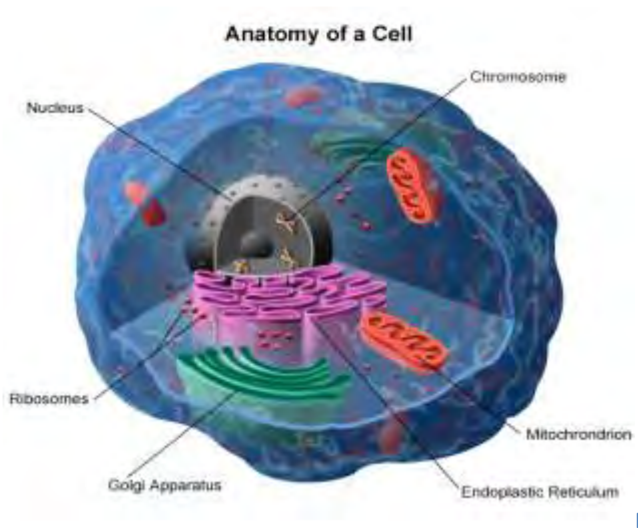


Figure 1

A prokaryotic cell. All the chromosomes of a living organism are stored in the Nucleus of that cell.

The nucleus of each cell includes all the chromosomes of the organism. The genome, as mentioned, contains all the genetic information stored in the DNA sequence derived from the cells in the way of chromosomes.

A chromosome is an organized structure of twisted DNA and protein organized in a three-dimensional structure. A discrete part of this twisted DNA that a chromosome contains is called gene. A gene is a molecular unit of heredity of a living organism. The DNA stored in a gene is used to create proteins that give the characteristic of every individual.

In other words, the genome of an organism contains both genes and sequences that apparently don't have any function, called junk DNA. Junk DNA is also the part of the DNA that is useless to create proteins.

Genomes according to their size (number of base pairs) are divided in 3 main categories :

Small Genomes: Bacterial genomes containing few Megabases,

Medius Genomes: Lower plant genomes containing hundreds of Megabases

Big Genomes: Plants and mammalian genomes: containing Gigabases

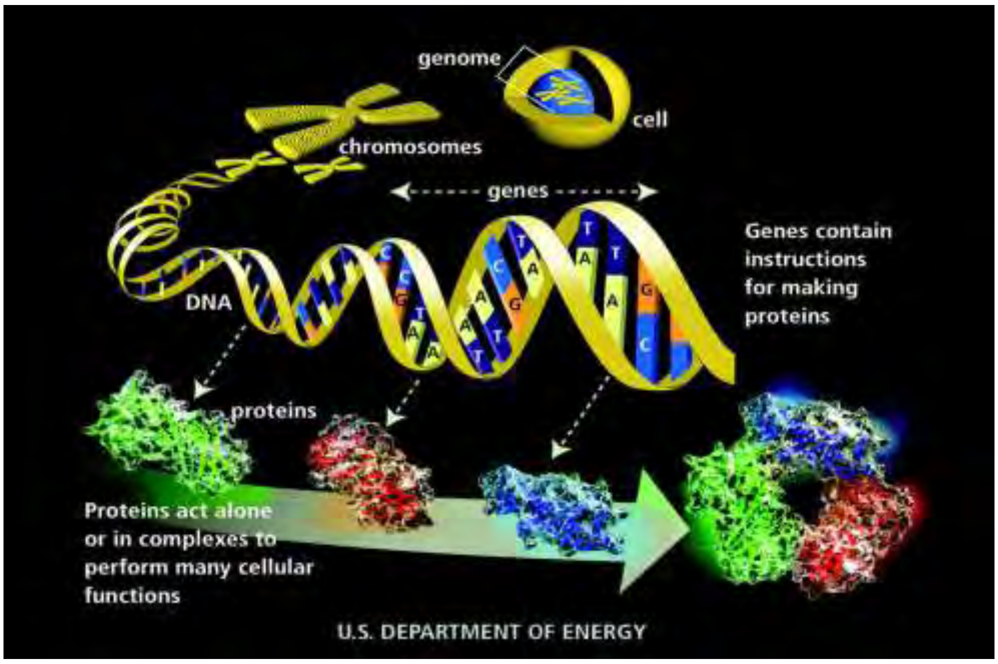

genome
cell
chromosomes
genes
Genes contain instructions for making proteins
DNA
T
A
C
G
proteins
Proteins act alone or in complexes to perform many cellular functions
U.S. DEPARTMENT OF ENERGY

Figure 2

Above is shown the nucleus of the cell with the chromosomes. Below is the coiled DNA of a chromosome that is divided indiscrete parts called genes. The genes contain the information for making proteins that distinguish every living organism. (Genome n.d.)

1.2 What is Genome Assembly?

Scientists made significant efforts to create methods that will determine the exact sequence of the genome in a living organism. These efforts led to the development of a very interesting sub-field of computation biology called Genome assembly. Scientists dealing with this field focus on building algorithms and tools that will solve the genome assembly problem.

All the current solutions to this problem are based on the same process. They start from multiple segments of the DNA molecule contained in every chromosome. Then these segments are randomly broken in a big number of short sequences, called reads, with the help of a shotgun sequencer. A shotgun sequencing project is the process of breaking the genome of an organism into multiple fragments of a generally small size. A more extensive explanation of this process is give in the next paragraph. The goal-job of a genome assembler is to stitch together the data, produced by the sequencer, in a correct way in order to form the sequence of the genome. In order to achieve this merge every read is compared with each other, aligned to one another, finding all places that two or more reads overlap. The reads that overlap can be merged together and the procedure continues like this till the whole genome is created. The outcome of an assembly is a collection of big sequences of the genome that are put together correctly.

Genome assembly it's a hierarchic structure: the reads forms the contigs, the contigs form the scaffolds.

- A read is a sequence determined by the sequencer.

- A contig can be seen like a multiple alignment of reads ,is the part that they overlap without a gap

- A scaffold (or supercontig) is an oriented and ordered sum of two or more contigs. Usually it contains gaps that are created by errors during the assembly process.

The assembly procedure is commonly referred as the process of solving a jigsaw puzzle. First we put every piece next to its other to check if they fit together and then we put the bigger pieces that are created into a place.

The difficult part of this process, even its not visible now, is that the genome contains over 30 percent of sequence that is repeated lot of times and belongs to different places of the genome sequence. Thus a repeat overlap can happened between fragments that are placed in totally different parts of the genome, thousands or even millions of base pairs apart. Gaps between scaffolds in the output final sequence are a result of these “wrong” overlaps.

Due to these repeats, the complexity of the examined genome and innate sequencing errors, produced by the using sequencing technology, the outcome of the first assembly run are rarely similar with the expected genome. Genomes that contain at most 1 error per 10.000 bases are considered finished genomes. These genomes are called drafts. Fixing the erros is a very difficult work that requires lots of hours and it also very expensive. Because of thsi lots of genomes are never fully assembled and they remain in a draft form.

1.3 Genome Sequencing

Sequencing is the process of reading and decoding the nucleotides of DNA or RNA composing the genome of a living organism. In this process the entire DNA is first isolated from the target organism. As sequencers have the limitation that they cannot take as an input more than a certain number of base pairs the extracted DNA must be fragmented. This work is made by a shotgun sequencer. In a shotgun sequencing project, named like this due to the quasi-random firing pattern of a shotgun, the extracted DNA is randomly broken into millions of small pieces which then are read by the sequencer and converted into digital information that can be stored in a computer. The results obtained from the sequencer are called reads. Multiple reads of the same DNA are generated by performing several rounds of segmentation and sequencing. Then algorithms use the overlapping parts of the different reads to assemble them into a continuous sequence.

The creation of multiple reads from the target DNA mentioned above is done to achieve high coverage. Genome coverage is a very important parameter in the assembly procedure. Coverage refers to the average number of sequences that independently contain a certain nucleotide. Every sequencer produces error but with different rates. These errors can be a skip of a base, a misread of a base or both. However, using high coverage allows a computer to determine the correct genome sequence based on consensus of the majority of the reads.

```
  ATGGCATTGCAA
   TGGCATTGCAATTTG
AGATGGAATTG
 GATGGCATT GCAA                Reads
     GCATTGCAATTTGAC
  ATGGCATTGCAATTT
AGATGGTATTGCAATTTG


AGATGGCATTGCAATTTGAC           Output
                               Sequence
```

Figure 3

The majority of the sequence determines the consensus sequence. In this example the sequences determine T as an error as far as C appears more times

A generalization of what is mentioned above is that high quality and accurate assembly is a product of high coverage. Parts of the genome in that a big number of reads overlap are said to have high coverage and the consensus is more reliable and parts that few read overlap are said to have low coverage and the consensus is not that reliable.

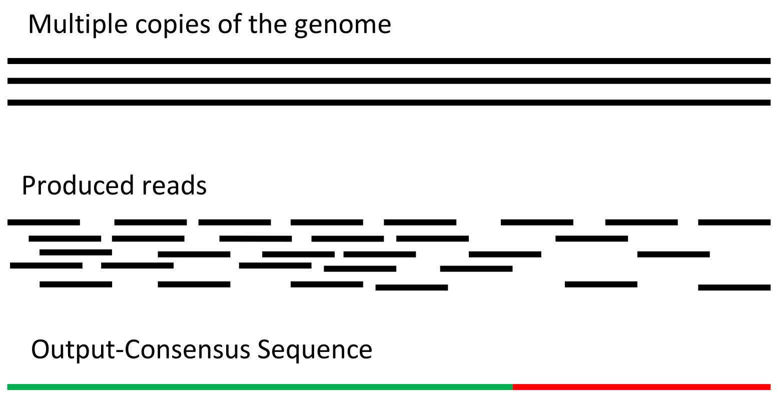
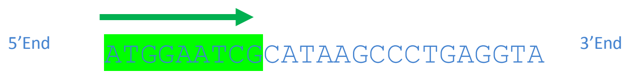


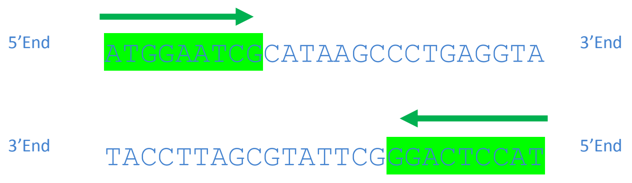
Figure 4

The multiplication in copies of the examined genome provide coverage. The output-consensus sequence is more reliable in the part that more reads overlap.

The procedure of sequencing a fragment of DNA or RNA can be done in two main ways. Sequencing only one helix of DNA (or RNA) from the 5’ end is called single-end sequencing (PE) and sequencing both helix of the DNA (or RNA) from their 5’ ends is called pair-end sequencing. Mate-pair sequencing is similar to pair-end sequencing with a difference in the size of the reads that are produced. Mate paired reads are usually much longer than pair ended reads.



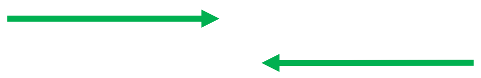
Single End (SE) sequencing of DNA fragment



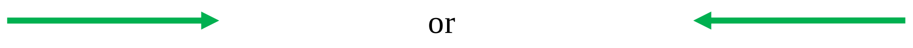
Pair End (PE) sequencing of DNA fragment in both stand

The sequencing technology provides another important parameter that is the orientation of the pair end (or mate pair) reads. Every orientation has its own meaning. The orientation that the reads have can be summarized in the following categories:

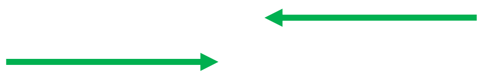
1. Left Right (LR). This orientation, that can be also found as Forward Reverse (FR) or (+/-), means that the reads are obtained from opposite DNA stands and due to this the left read is forwad and the right read reversed. Libraries with this orientation are often called pair end libraries with Sanger format.



2. Left Left (LL) / Right Right (RR). Reads oriented in this way, also named as Forwad Forward (FF)/ Reverse Reverse (RR) or (+/+) /(-/-) respectively, are obtained for the same DNA strand and due to this they have the same orientation. The method that produce read from the same strand is known as circularization. Some assemblers, like Velvet, expect paired-end reads to come from opposite strand facing each other. Due to this if circularized reads are going to be used the first read in each pair must be replaced from it reverse complement before starting the assembly process.



3. Right Left (RL) . named also as Reverse Forward (RF) or (-/+), are obtained from different strands but in the opposite way of the LR pair ended reads. In order to uses these libraries with assemblers that expects traditional Sanger format, like Velvet, both reads in each library must be substitute by the reverse complement before starting assembling them. The mate pairs library used in this study has this orientation



Pair-ended and mate-pairs reads provide to the assembly procedure constraints that define the location of the reads. These constraints lead us in a decrease of the ambiguous regions and because of this longer genome scaffolds are created.

1.4 Assembly Categories

Two are the main categories that the assembly of a genome can be distinguished. These are:

1. De-novo: the reads are assembled to form a sequence not known a priori. Sequence reads are assembled in longer contiguous sequences called contigs, followed by the process of ordering contigs into scaffolds without the help of a reference genome.

2. Mapping/reference: consists in assembling reads based on an pre-existing genome sequence. This sequence is used to align the reads of new genome avoiding the process of creating data structures as with the de novo assembly. The generated sequence is similar with the backbone sequence but not identical.

The de novo assembly problems from the mathematical point of view can be categorized as NP-Hard. NP-hard problems don't have an efficient computational solution, so they are harder to be solved from the other problems. Even if de novo assembly problems take longer time to be solved and they need more computational resources they are the only way to assemble genomes that are not suitable with a reference one. In the present study we will focus in the category of assemblers.

On the other hand, mapping assembly is faster and need less computational resources compared to de novo assembly. Also mapping is a much easier way to assemble because it is sufficient to align the read to the reference genome

1.5 Why is Genome Assembly important

The genome sequence can be seen like a treasure for the scientists. The wealth of these data led to a big amount of discoveries in the field of biology. These discoveries help to understand many principles of life.

By knowing the genome sequence scientists can find genes faster and easier due to some clues that the sequence contains. Scientists can also understand how the whole genome works. More precisely how genes work together to direct the evolution and the maintenance of an organism. Moreover, because the genes cover the only the 25 percent of the genome scientists can study more deeply parts outside the genes. One of these parts can be the stretches of junk DNA that till now is believe that it doesn't have any biological function.

2. Assemblers

2.1 Introduction

First assemblers, developed in 1980s, were using longer fragments that allow better identification of the sequences that overlap. In this generation of assemblers, called first generation, fragments-reads were obtained by Sanger sequencer that can produce read up to 1000 bases pairs. Sanger sequencing is based on the chain termination method using capillary electrophoresis. In this technology problems were created because the algorithms that were using long reads show quadratic or even exponential complexity behavior. Also the high cost and the slow throughput of Sanger sequencing made it unsuitable for sequencing whole genomes.

As the year passed sequencers, aiming on high coverage, became faster and with a lower cost they could produce reads that were shorter forming a new generation of sequencing called Next Generation Sequencing (NGS). The ways of detecting overlap and building the contigs are the same for an assembler regardless of the read's length. In practice although, using the existing -first generation- assemblers with short read failed for a variety of reasons. Some reason were based on the algorithmic part of the existing assemblers as these assemblers impose a minimum read length or they require a minimum amount overlap that is too long when short reads are used. Another reason was that the part of the algorithm that have to define the overlaps between the reads is one or maybe the most critical step in an assembly process. Short read sequencing algorithms require a redesignation of this step to make it feasible especially since many more short read are needed to achieve the same level of coverage. Coverage used with the current technology usually ranges from 30X-50X while with the first generation were limited at 8X.

Mainly for these reasons, a new generation assemblers has been developed. While short reads are faster to align they also created some problems. The major one is that shorts reads are more difficult to use with repeats or near identical repeats. Moreover, short reads make the NGS platforms not particularly suitable for the sequencing of new genomes, especially of big dimension and rich in repeats genomes. Although nowadays short reads assemblers are the most used to assemble and they are able to assemble most of the genomes.

The complexity of an assembly procedure is mainly based on the number of reads and their length. More reads achieve better coverage and and longer reads achieve better overlap. Due to these factors in the last years scientists are trying to find a middle way between first and next generation assemblers in order to use longer reads, compared to the one currently used, keeping the benefits of the current technology.

The main sequencers that are currently used to create short reads reads are Illumina Solexa, Roche's 454 Life Sciences, Applied Biosystems' SOLiD systems and Ion Proton from Life Technologies. They can produce 50-1200 base pairs depending of the technology that is used. However, reads produced from these technologies are less confident because error rates in these generations are higher. Although, the high speed and the low cost can solve this problem with redundant coverage, as a nucleotide can be sequenced many times. A summary of every technology is shown in the next table.

Summarizing, we can say that the main differences between the first and the next generation sequencers are the lower cost and the higher throughput of NGS but with the disadvantage of higher error rates and shorter read lengths.

From 2007 several large genome have been published that use a combination of the first generation and NGS. An initial assembly was created with a first generation Sanger data and Next generation sequence data were used to fill the gaps.

Technology	**Sanger**	**Illumina Solexa**	**Roche 454**
Sequencing Machine	3730xl	HiSeq 2000	GS FLX Titanium XL+
Sequencing Method	Dideoxy chain termination	Sequencing by Synthesis	Pyrosequencing
Time per run	~23 Hours	~ 11 days	23 Hours
Mb (Mega bases) per run	1.9~84Kb	600 Gb	700 Mb
Read Length	400-900 base pairs	100 base pairs	700-1000 base pairs

Cost per Mb	$ 2400	$ 0.03	$ 84.39
Accuracy	99.999%	98%	99.997%
Instrument Cost	$ 95,000	$ 690,000	$ 500,000
Strengths	+ Long read length + High quality throughput	+ High throughput (highest one) + Low cost per base	+ Long Reads + Fast sequencing executions + Handles well GC regions
Weaknesses	− Low throughput − High cost per Mb	− Handles bad AT- and GC-rich regions − Error rate at 1%	− Low throughput − Error rate 1%

Technology	**SOLiD ABI**	**Ion Torrent**	**HeliScope**
Sequencing Machine	5500 series	Ion Proton Sequencer	tSMS
Sequencing Method	Ligation-based sequencing	Ion semiconductor sequencing	Single molecule sequencing
Time per run	9 days	2 Hours	7 days
Mb (Mega bases) per run	170 Gb	Up to10 Gb	21-35 Gb
Read Length	35 base pairs + 75 base pair	Up to 200 base pair	25 base pair + 55 base pairs
Cost per Mb	$ 0.004	$ 4.85	$ 0.005 (per base)
Accuracy	Up to 99.99%	99.6%	99.995%
Instrument Cost	$ 595,000	$ 149,000	$ 999.000
Strengths	+ Low cost per base + High throughput + If reference genome is available provides high accuracy	+ Fast throughput (fastest one) + Low cost instrument	+ Library preparation easy
Weaknesses	− Handles bad AT and GC rich regions − Sequencing times usually long − Most of the times doesn't work with colour space	− High cost per base − Doesn't prone homopolymer errors	− High error rates

In the present study we will focus on the this new generation of genome assemblers that use short reads in order to test them their efficiency with different genome's size in a big machine. The assemblers that are going to be tested are Velvet and ALLPATHS-LG for shared memory and ABySS and Ray for distributed memory. These assemblers have in common not only that they belong in the new generation of genome assemblers but also that they are graph based. Graph based assemblers are the most used and successful in achieving decent results with all the genome's sizes. As mentioned, when a reference genome is not available the only way to assemble a new genome is by using de novo assembly. The de novo genome assemblers can be divided in three main categories:

- Greedy assemblers that use is an implementation of string-based method

- Overlap-Layout-Consensus (OLC) assemblers and
- De Bruijn Graph assemblers (DBG) that both use graph based approaches.

A short description of these three categories in given in the next paragraphs.

2.2 Assembly Errors- Short Description

Before introducing the graph based de novo assembler categories a short description of the errors that can be localized in graph that they create is given.

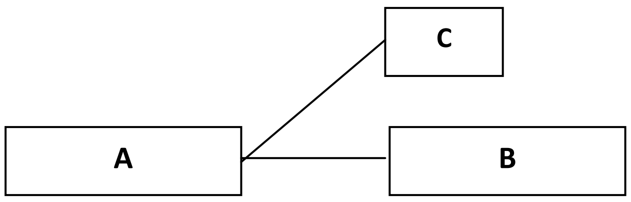
Graph assemblers as all the assemblers don’t reconstruct the 100% of the target genome because errors can affect the procedure. These errors can occur for two basic reasons:

1. Incomplete and/or incorrect data that are given as an input to the assemble
2. Limitations on the assembly process

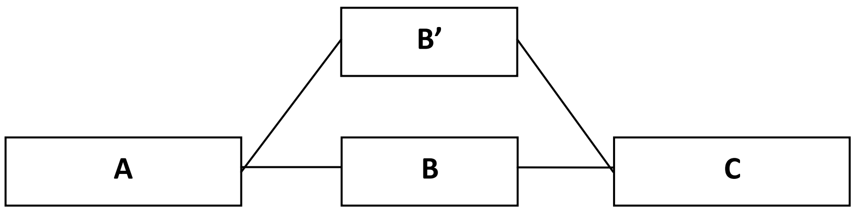
The problems that can be errors create in a graph (overlap or de bruijn graph) are presented below:

- Spurs: short dead end branches of the main path. They are usually caused due to sequencing errors in one end of the read and low coverage
- Bubbles: deviation of the main path in two branches that after rejoin together in one path. They are usually caused due to sequencing errors in the middle of the read and by the complexity of the target genome.
- Paths that converge into one and after they separate again into two distinct paths. They are usually caused due to repeats in the target genome.
- Cycles: Paths that created a loop on the main path. They are usually cause due to repeats in the target genome
- Chimeric Connections: Connection of genuine contigs in “non-legal” way. These connections can occur for a random overlap of two tips that belong to different parts of the graph or for an erroneous alignment of a read with another part of the genome.

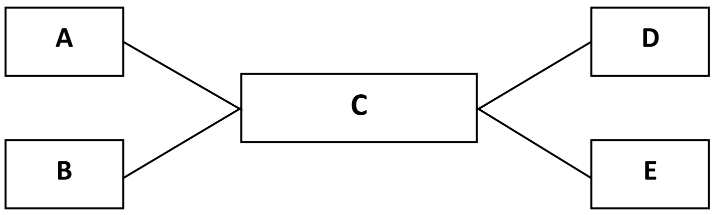
A)



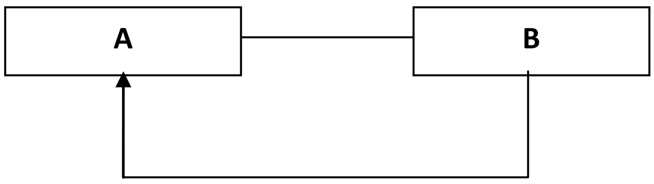
B)



C)



D)



E)

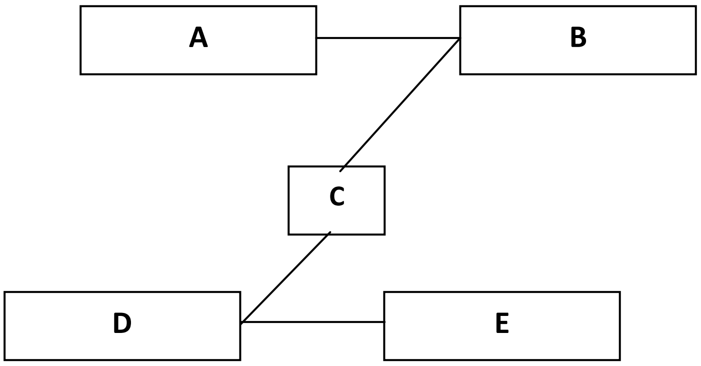


Figure 5

In this figure is shown the structure of the assembly errors that can be encountered in a graph: A) A tip, B) A bubble, C)A converging and diverging path, D) A cycle and E) A chimeric connection

2.3 Greedy Graph Assemblers

Greedy assembly algorithms were the first de novo algorithms that appeared. Their main work is to calculate pairwise alignments of all the reads provided as input. Then, these alignments are scored with grades that represent the length of the overlap and the percentage of matching bases. The two reads with the highest grade are merged together creating a contig. The created contig is placed with all the rest of the reads in a "pool" of sequences. The operation of extending sequences from the "pool" continues till no more quality overlap exists.

The assemblers of this category simplify the graph by passing from the high grade edges. Missassemblies are tried to be avoid with mechanisms that terminate the extension of the sequences when information that casues conflict is found. This information consists in overlaps that two or more reads have with the same contig but they don't overlap between them.

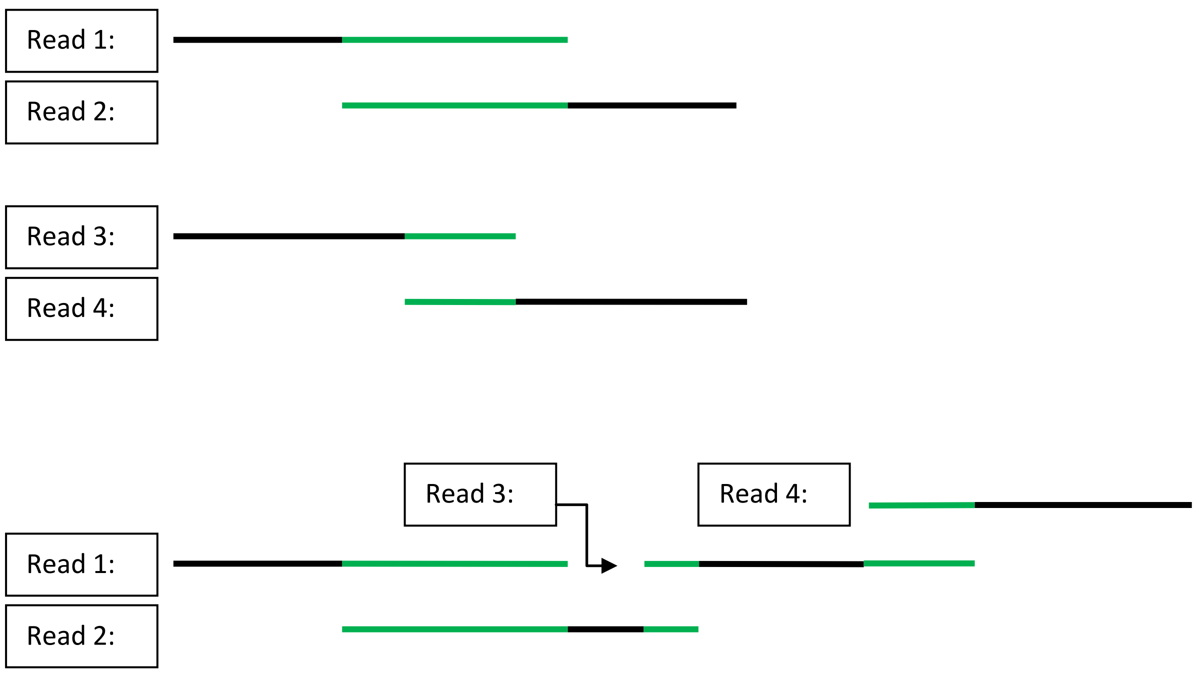


Figure 6

An example the greedy approach. The assembler merge first reads 1 and 2 because they overlap in a bigger area than reads 3 and 4 and reads 3 and 2. Then merges reads 3 and 4 with the second bigger overlapping area and finally 2 and 3, that they have a small overlapping part. A contig is created in this way using only local information.

The greedy algorithms although can stuck at a local maxima if a contig is extended with a read that would help other reads or contigs from the pool to grow even larger. In general the basic disadvantage that they have is consider only local information at each step so the assembler can be easily cheated by complex repeats that will lead to mis-assemblies. Assemblers in this category are also memory intensive making them inappropriate big and complex genomes.

Some assemblers that belong to this category are:

- TIGR
- CAP3
- SSAKE
- SHARCGS
- VCAKE

2.4 Overlap-Layout-Consensus Assemblers

These assemblers, mainly developed in the first generation of assemblers, operate in three main phases and are more suitable for long reads. Every OLC assembler uses this 3 phases with a different approach and that’s why in this category there is a big amount of different assemblers. The three main phases are the following:

Overlap Phase: In this step every each read is compared with every other to find the area that they overlap and an overlap graph is constructed with the information provided by the comparisons. Each read is represented by a node and an edge between two nodes shows that these reads overlap. The creation of an edge between nodes depends on the assembler’s tactic. The most used tactic is to create an edge if the reads-nodes overlap with at least K bases of a Y% similarity. This method makes overlap computation a very time intensive step- especially if the set of read is very large.

Layout Phase: In this step the graph, constructed in the previous phase, is analyzed and simplified, with the application of graph algorithms in order to identify the paths that correspond to segments of the genome sequence. These paths are made by reads that they overlap and form contigs. Contigs in this approach form subgraphs that contain lots of edges between the node that this subgraph contains. The simplification of the graph consist also in merging the nodes of the subgraph in unique. The ultimate target is to find a Hamiltonian path (i.e. path that traverses each node in the graph exactly once) that will be considered as an approximate layout of the reads in the final genome sequence. The procedure of identifying a Hamiltonian path is a NP-complete problem for which the time required to solve it increases exponentially with the problem's size.

Consensus Phase: After the previous step the consensus sequence is derived. The graph is reduced in large scaffolds with application of alignments in the sequences. They ideal outcome is a single scaffold with not gaps, but as repeats and inefficient information cause problems in the algorithm the final output consists in multiple scaffolds with gaps between.

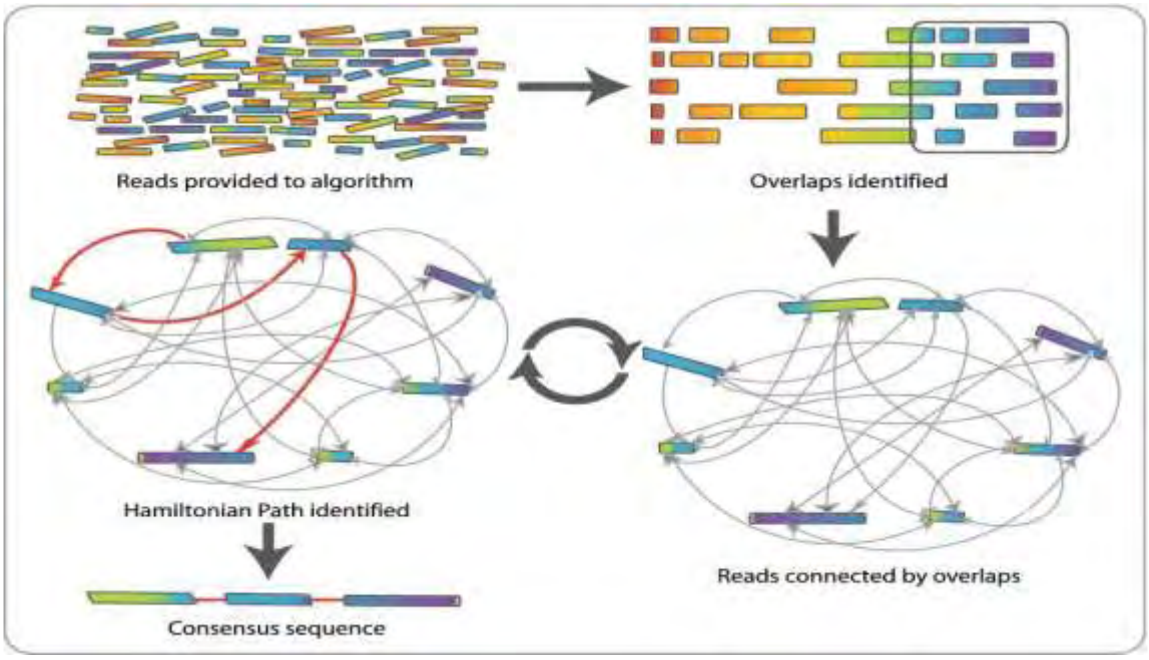

Reads provided to algorithm
Overlaps identified
Hamiltonian Path identified
Reads connected by overlaps
Consensus sequence

Figure 7

The OLC process. The reads that are provided to the algorithm are examined pairwise for overlaps. The graph is created with the reads that represent the nodes and the edges the overlaps between them. Then the algorithm searches for the best Hamiltonian path in the graph. Unused nodes and edges are removed. The procedure is repeated many times and the resulting sequences are combined to achieve the consensus sequence that will represent the genome. (Jennifer Commins 2009)

Some assemblers of this category are:

- PHRAP
- Celera
- Arachne
- Phusion
- Euler

Advantages-Disadvantages

The major advantage, due to the three phase implementation of OLC assemblers, is that an optimization and a modification in order to make these steps more efficient can take place. Any step can be improved independently from the others and driven to handle the needs of the assembly procedure. Another advantage is the OLC assemblers can use data either from the first generation sequencer, like Sanger, or from NGS platforms because the overlaps among the read can vary in length.
On the other hand, the cost of the overlapping step is very time intensive as in this phase each read is compared to every other to determine the overlaps. OLC, as mentioned, can use NGS data but because this data consist in a bigger amount of reads, a significant increase of the overlapping step will be caused. Moreover, finding a Hamiltonian path (i.e a path that traverses every node just one time) in the layout phase is an NP-problem not solvable is not yet polynomial time and this makes OLC dependent of heuristics in order to achieve more confident results.

Generally OLC assemblers are consider being inappropriate for NGS. The major reasons are:

1. The overlap graph (a node per read) becomes extremely big and "heavy" to calculate
2. The small dimension of the reads produces lots of ambiguous connection in the graph
3. Many algorithms require a minimum overlap that is superior to the length of the reads obtained from NGS
4. The big number of reads, the short overlaps and the high frequency of sequencing errors create problems in the execution in the different phases of the algorithm.

2.5 De Bruijn Graph Assemblers (DBG)

Nowadays the most utilized approach in combination with NGS data is the de Bruijn Graph approach. The first application of de Bruijn graphs for the assembly of a genome was proposed by Pevzner in 2001 and is currently there is big amount of assemblers base on the approach to handle short read data produced for NGS platforms. The backbone steps are described below. Although every algorithm adapts these steps to its needs.

Calculation of the k-mer and construction of the De Bruijn Graph

In order to build the de Bruijn graph, the assembler divides all the reads in overlapping segments of length k that are called k-mers. Then uses these k-mers to build the de Bruijn graph. In a general way, we can say that the nodes represent the k-mers and the edges the overlaps between them. The exact way of building the graph differs in every algorithm and will described in detail in the algorithms that will be presented in this thesis. The way of using k-mers to build the graph decreases construction time because no pairwise overlaps are calculated. Every k-mer is stored just one time in the memory despite the times that appears in the genome that makes the construction of the graph easy with the use of a hash table. So high redundancy does not affect the number of nodes.

Error Correction

The de Bruijn graphs that are created in the previous step are, usually, very sensitive to errors, especially to sequencing ones caused by the introduction of k-mers. As the reads are divided in k-mers new sequencing errors can occur. Some assemblers preprocess the reads so these new sequencing errors that occur in the k-mers are avoided and then they create the graph, so we can say that the error correction is included in the graph construction step, and some other assemblers use methods that identify and correct the errors by examining the graph structure. The errors that can be found in a graph are described in 2.2 paragraph. These erroneous graph structures are localized, corrected or removed from the graph.

Scaffolding

In this step every assembler uses information determined by its implementation, like pair end sequences, clone maps, restriction maps, mate paired sequences and also, but not so often, information from a related genome to gather, orient and glue segments.

Usually, scaffolding is based on mate-pair information. Two contigs can be merged to one if one end of a mate-pair is contained in the first contig and the other end is contained in the second contig. Although, in practice two or more mate-pairs are required between the two contigs in order to avoid experimental errors. Often greedy approaches are used in order to create scaffolding techniques. These techniques begin by using the most reliable information and then they incorporate data. This procedure is done in a loop that ends when a new information create a problem with the already build sequence.

Finishing

As mentioned, the assemblies that end in scaffolds separated from gaps are called draft assemblies. In these assemblies the process of filling the gaps to obtain the final sequence is called finishing. This step fills the gaps, improve the low quality regions, resolve missamblies and orders scaffolds. The first sequencing leads us, most of the time, to draft assemblies, so a second assembly of the whole DNA is done to fill the gaps. Also, sometimes, instead of reassembling the entire DNA an amplifying and sequencing of the segments that end to gaps is performed. There are several tools to perform this final step like IMAGE.

Some assemblers of this category are:

- Velvet
- ALLPATHS and ALLPATHS-LG
- ABySS
- Contrail

Advantages-Disadvantages

De Bruijn assemblers greater advantage in comparison with the OLC approach is that no computation of the pairwise overlaps is done. As mentioned in the OLC approach this computation is very time expensive and the things get worst when large data sets are used. K-mers are shorter reads and they are stored just one time in memory a thing that allow to build the graph easily and without memory consumption. The choice of the k-mers is very crucial in this approach: Small k-mer length increases the connectivity but also increases the ambiguous region while big k-mers increase the specificy but also decrease the connectivity.

This approach has a k-mer centric nature meaning that its topology is unaffected by the fragmentation of the reads. This makes de Bruijn assemblers efficient for comparative genomics or when mixed length reads are used. Moreover, the one to one relationship between path and sequences that the de bruijn graph make the overlapping sequences follow the same path.

An additional advantage of this approach is that the topology of the graph allow to treat efficiently the error correction step. Repeats and sequencing errors are easier to recognize and removed or correct than in an overlap graph.

Finally, the structure of these graphs makes them efficient for cluster memory distribution and multithreading in shared memory approaches. This advantage led in the creation of multithreading version of some assemblers, like Velvet, and of parallel assemblers like ABySS and Ray.

On the other hand, the sensitivity of this approach in sequencing errors increase the complexity of the graph as more nodes and edges are created. Furthermore, the use of Sanger data that is long lead to a loss of information. As long read are sequenced in k-mers a loss of the long range connectivity occurs. This loss of information can generate ambiguous region in the graph caused by short repeats and an accumulation of false positive overlaps can take place. Even if the identification of pairwise overlaps is time and memory intensive can be extremely useful to determine if the two overlapping reads come from the same genomic locus as this method is free from loss of information.

Moreover, as many eulerian paths can be located in a graph, every assembler using this approach has to imply some constraints in order to identify the original genomic sequence. The constraints, although, can lead in a creation of NP-hard problem instead of a polynomial one.

3. Algorithmic Description

3.1 Introduction

In this study two algorithms are going to be examined in order to analyze their behavior in a cluster machine. These two assemblers are Velvet and ABySS. Their main difference lies in that Velvet is built for shared memory distribution while ABySS for distributed memory. Velvet supports OPEN MP in order to use multithreading while ABySS uses MPI to pass messages between the nodes that it uses. Their common point is that they belong both to the de novo graph approaches that were described previously. They both use de Bruijn graph, built with a different approach in each assembler, and they both use the graph to identify and correct errors. Their algorithmic steps will be described in the next two paragraphs.

3.2 Velvet

3.2.1 Introduction

Velvet is a set of algorithms written in C programming language created to manipulate de Bruijn graphs for genome assembly. It implements a graph structure slightly different from the one proposed be Pevzner in 2001. It’s a suitable assembler for short reads, ranging from 25-50bp with a high coverage. The use of de Bruijn graphs help to remove errors produced by the sequencing machine and resolves repeats caused by the complexity of the genome. The removal of the errors and the removal of the repeats are done in two different steps. First an error connection step is applied and connects sequences that can be merged without ambiguities and then a repeat solver algorithm takes place to distinguish paths that locally overlap.
Velvet is divided in four steps: It has the reads into k-mer, it constructs the graph, it corrects erros and finally resolve repeats. These steps have different

computational requirements with a main bottleneck in terms of memory and time on the graph construction step.

3.2.2 Velvet's de Bruijn Graph description

Even if the basic idea of the graph structure,as mentioned, was based in Pevzner's (2001) implementation, the graph built by Velvet present some differences. In Velvet's de Bruijn graph a node N is created for a series of overlapping k-mers. Neighbouring k-mers overlap by k-1 bases. The last base of every k-mer that belongs to that series is its marginal information. The sequence of these last bases of each k-mer in a node is called sequence of the node N and its represented as s(N). Every node N its glued with Ñ that is its reverse complement of k-mers present in N in a reverse order. In this way the graph created is a bi-graph. This is done in order to make sure that the overlaps in the other strand of DNA(or RNA) are taken into account. An directed edge is create between two nodes if the last k-mer of the node that the arc starts overlaps by k-1 bases with the first k-mer of the destination node. Due to symmetry, if an edge is going from node N1 to N2 then there is an edge going from Ñ2 to Ñ1. Any change applied to a node or a edge is applied also to its complementary node or paired arc. With this structure reads are "translated" as paths in the graph.

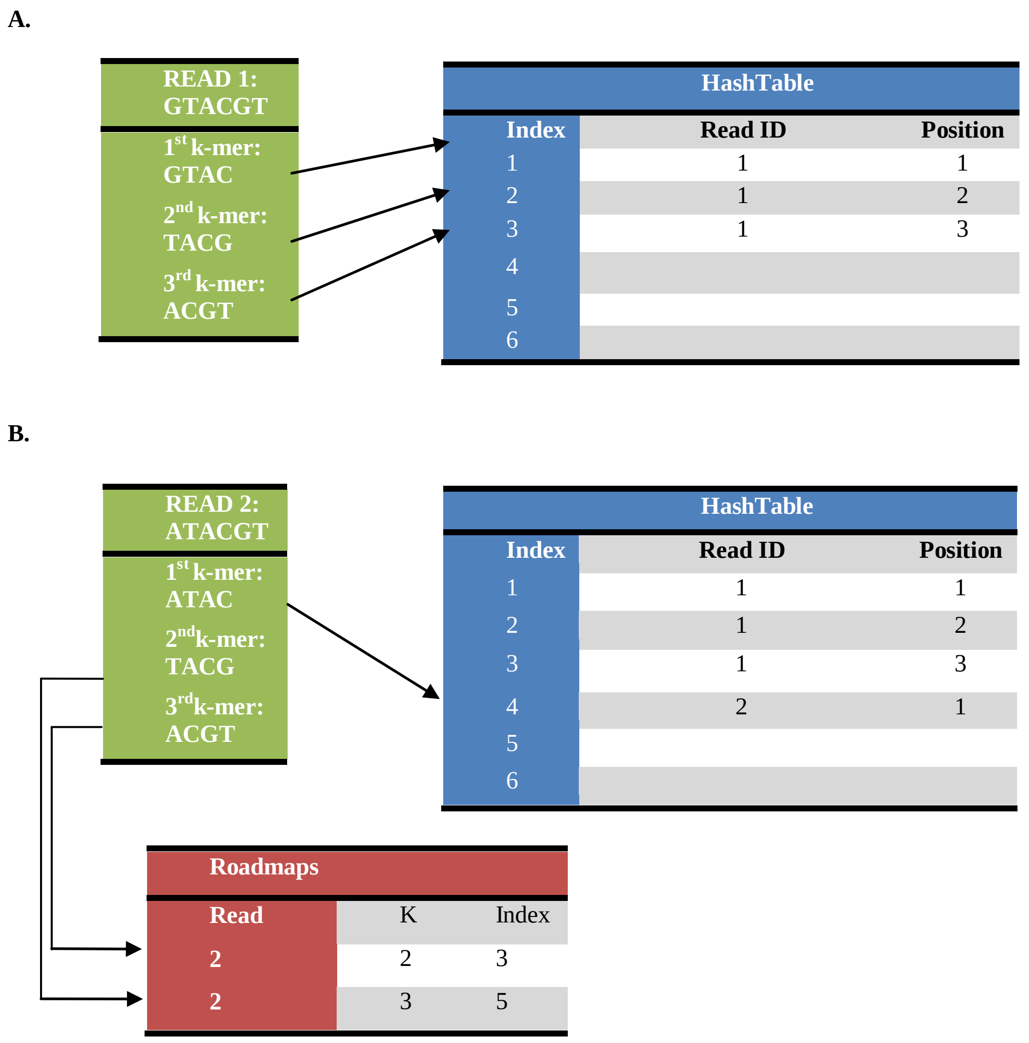
3.2.3 Velvet-Assembly procedure

Graph Construction

The reads given as an input to the assembler are first divided in k-mers whose length is defined by the user. The k-mer's length determines the quality of the assembly. In order to have decent results k-mer lentgth should be smaller than the reads length. In this way more overlaps among the k-mers will be observed. Generally, a k-mer with length near to the reads length creates a small amount of overlaps while smaller kmers increase both the chance of overlaps between kmers, caused from errors, and the ambiguous repeats formed in the graph. Moreover, smaller kmers increase the connectivity of the graph. The choice of k-mer length depends mainly on the coverage of the reads and the complexity of the genome. Different k-mers should be tested in order to find a k-mer length that will lead to decent results. For short reads with 25bp usually a good k-mer length is k=21bp. An important observation here is that only odd lengths can be used in order to avoid the case that a k-mer is its own reverse complement (for example k-mer ATAT). Such a case is a problem for the current de Bruijn bi-graph implementation. Although if the user tries to use an even length, Velvet will decrease and the start its execution.

Velvet graph construction procedure start with the procession of input files that contain a huge amount of reads. These reads are scanned, converted in an internal format and then saved in a file called Sequences. Then k-mers are produced from reads. Velvet creates a hashtable of n entries and every time that a k-mer is observed the algorithm search for it in the hash table. If is not present the hash table stores the ID of the read that this k-mer belongs to and the position of its occurrence within that read while if is present a reference of this k-mer is stored in another file called Roadmaps. In other words, Roadmaps file, determines for each read which k-mer was seen in a previous read. The hash table is temporarily saved in the memory while Roadmaps file is saved permanently.

When all reads are scanned the hash table and the Roadmpas file are used to build the graph. Each k-mer which "belongs" to a read (i.e. never seen previously) is a node. Each read is a path through the nodes/k-mers created by that read or previous reads (cf the Roadmaps file). This path is created by adding edges between the nodes.



HashTable (A)

Index	Read ID	Position
1	1	1
2	1	2
3	1	3
4		
5		
6		

HashTable (B)

Index	Read ID	Position
1	1	1
2	1	2
3	1	3
4	2	1
5		
6		

Roadmaps

Read	K	Index
2	2	3
2	3	5

Figure 8

A. No changes in the Roadmap file. The k-mers are loaded in the hash table

B. K-mer 2 and 3 where already present in the hash table. A reference in Roadmpas file is creat

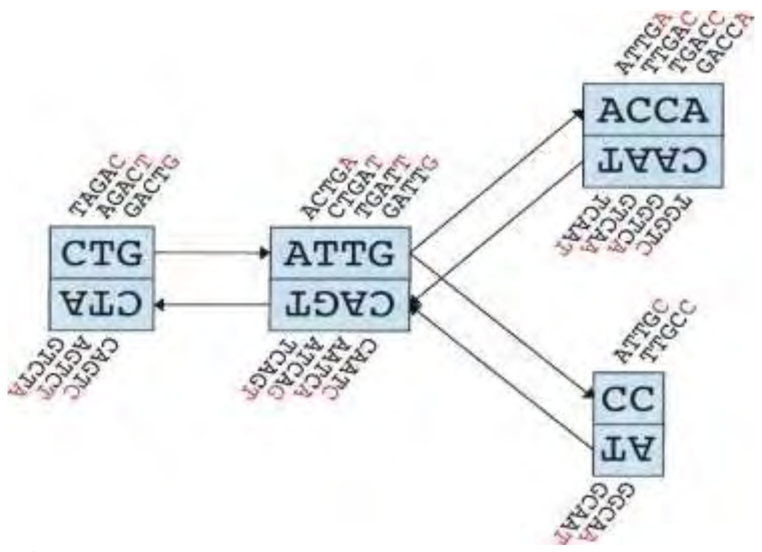


Figure 9

Example of a Velvet's de Bruijn Graph

- Graph Simplification

Usually after the construction of the graph there are a lot of simplification that could be done between the nodes as a node is created for every k-mer. This method create in the graph chains of nodes that can be merged saving both memory space and computation time without losing any information. The merging process take place when a node N1 has only one directed outgoing edge to a node N2 that has only one incoming edge. These two nodes can be merged in one containing all the information that the two starting nodes had. An example is shown in Figure 9

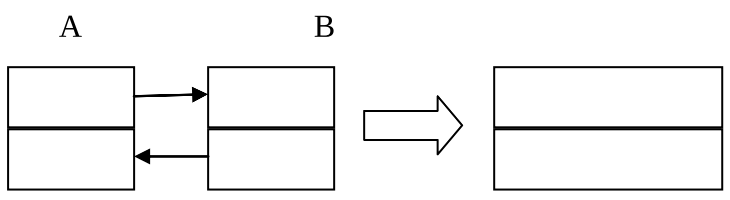


Figure 10

Simplification method results in the merging of two nodes.

- Error Removal

As Velvet focuses on topological features error removal take place after the graph creation. Errors, cause by the sequence technology or repeats in the target genome, form some typical structures in the graph that can be located and removed.

The most common error that can be located in a graph are spurs. As described they are short dead end branches that are separate from the main path extremely frequent when NGS data are used. Tips are formed when an error occurs less than k bases from the end or the start of the read. Deleting these branches has only local effect as the connectivity of the graph is not changed. Although lack of coverage can lead to spurs that are not errors and in order to avoid the removal of genuine sequences two parameters are taken into account. These are the length of the spur and the minority count.

A spur will be removed if its length is less than 2k bp where k is the size of the k-mer. Usually spurs longer than 2k bp are not erroneous sequences, and if they are they contain big amounts of errors that are difficult to distinguish from the correct sequences. In the second case it’s good if the user apply another value to the cutoff parameter.

Minority count has a spur when the node that starts the derivation has at least one edge with higher multiplicity than the edge going to the tip. This mean that the path that leads to the tip is made by fewer reads (or small coverage depth) than the other paths passing from the derivation node.

Using the the above two constraints applied tips are removed iteratively with an increasing order of multiplicity. This process removes tips without causing an erosion of the graph and reveals paths with higher coverage. When this process terminates a simplification step, like the one described before, is again applied in this “new” graph as nodes could be merged to improve the graph structure.

In a graph another error that can be easily identified are the bubbles, branches in the graph that have same start and end node but they differ in the middle, caused by error in the middle of the reads or k-mers, or from random overlaps of two nearby tips. They way that Velvet treat them is with the “Tour Bus” algorithm.

In briefly, this algorithm, executes a breath-first search in a Dijkstra mode. It start from a node and goes through the graph visiting nodes of increasing distance. The distance between two nodes N1 and N2 is the length of s(N2) divided by the multiplicity of the edge going from N1 to N2 (i.e divided by the number of reads going from N1 to N2). In this way priority is given to more secure-reliable high coverage paths. When the procedure founds a node that was already visited it backtracks to the closest ancestor. The two paths that end in the node that cause the backtrack are extracted aligned and if they are similar they are merged. The path that first arrived at the end node(that one that when was encountered the second time caused the backtrack action) is uses as a backbone due to its higher coverage.

Although when the merging takes place the outcome sequence have to relocate the connections with the nodes that the two paths were connected to. In linear paths, paths that don’t contain blocks visited more than one time is easy because the connections are simply repositioned in the merged sequence. In palindrome paths, paths that go through a block one way and then throught it in the opposite direction the merging process is more difficult as connectivity will be affected if they are treated like linear paths. Due to this the Tour Bus algorithm marks every node in the two branches and starts the merge procedure from an end to the other visiting nodes consecutively. Each node of the minority branch is compared with the corresponding node of the consensus sequence. All the information,like coverage, edges and sequence identifiers, that the minority node had is transferred to the consensus node. In this way the marked path is changed dynamically without loss of connectivity.

A.

B.

C.

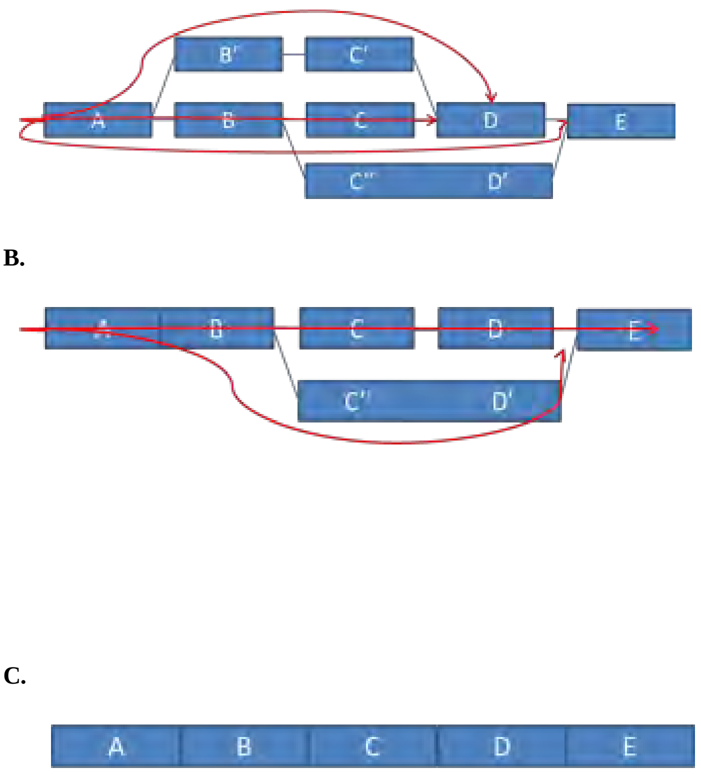


Figure 11

Example of Tour Bus execution.
A) The search start from A and goes toward the right.. The procedure that goes from B′ and C′ stops at D as this node was already visited. The nucleotidic sequences of B′ C′ and B C are removed from the graph, aligned and compared.
B) The two were consider similar and they are merged into BC. Then the procedure continues to the bottom path and going through C′ and D′ end in E. The paths C′D′ and C D are compared and merged as were considered similar.
C) The final graph.

After the Tour Bus algorithm long straight nodes are created that have high coverage. Moreover short nodes that couldn’t be simplified are low complexity sequences present a lot of times in the genome with an elevated coverage value.

So, short nodes with low coverage usually correspond to chimeric connections (i.e nodes which incorrectly connect two unrelated contigs). These connections should be removed to achieve better results. After Tour Bus a removal of these erroneous connections is applied to the graph. Velvet uses a coverage cutoff parameter, set by the user, to remove nodes with coverage less than this value. This parameter, although, should be set carefully because a high value can lead in gaps formation and missasemblies as genuine nodes will be removed. A tactic that is generally used to decide the value of this parameter is based on the observation of the coverage distribution of the contigs.

- Repeats Resolution

This step can be considered as the second phase of error removal. The first phase ends with contigs separated in branching points caused by repeats. This step aims in determining the genome’s sequence path that goes though the repeated nodes in the graph. A repeated node is the one that its sequence is present multiple times in the genome. If the repeat is longer than the sequence contained in a node then this repeat is a path in the graph from which the genome sequence pass multiple times. A simplified version of a repeat structure is presented in the next figure.

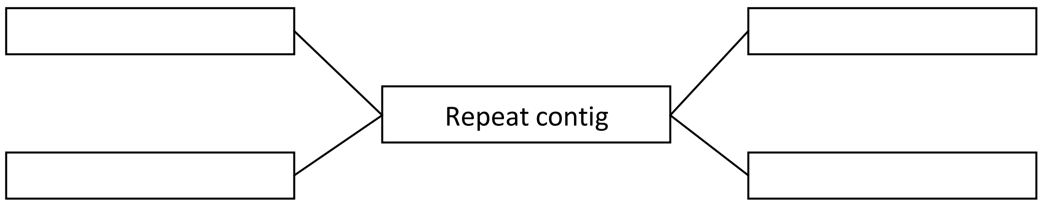


Figure 12

A simplified common repeat structue

- Resolution of repeats and scaffolding

Velvet's final graph reduction step involves mate pairs (if provided). An algorithm called Pebble is used to exploit paired end information to remove repeats and build the final scaffolds. The first that is made is to identify unique nodes. Velvet uses coverage to identify these nodes and then Pebble tries to connect them using pair end information. For every unique node, chosen in an decreasing order of coverage value, estimates the distance from this node to the next unique one and then merges the distance information that these two nodes provide.

In detail before resolving repeats Pebble build a primary scaffold. For any two nodes in the graph Velvet counts the mate pair reads that connect them. Then using a Maximun likelihood estimator finds out the distance between them. All the inter-nodes that define the distance between the two examined nodes is called primary scaffold.

Then in order to merge unique nodes it is needed to find out which nodes are in the neighborhood of every unique node. It uses the primary scaffold to find nearby nodes but as this information is not sufficient, because usually the given insert length is bigger that the examined unique's node length and this creates no primary information about the neighboring nodes, Pebble tries to compute the distances between the local nodes. This is called secondary scaffold.

What is done in an abstract view is shown in the figure. Primary neighbors are the unique nodes that are connected with the examined node. This information is provided by the primary scaffold. Then Pebble searches for all the directly connections of these nodes, flags the nodes that the end that they are called secondary neighbors. With the estimation of distance computed between the examined node and its primary neighbours plus the distance between these nodes and the secondary neighbors it is easy to find the distance between the examined unique node and the secondary nodes by doing a subtraction.

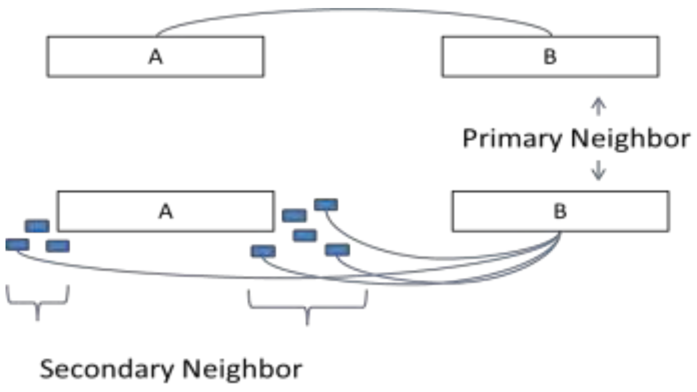


Figure 13

Pebble discovers all the unique nodes that an examined unique node is connected. Here unique node A is connected with unique node B. For every unique node B which is connected to A, Pebble then follows the primary connections associated to B, thus flagging secondary neighbors of A. Assuming that all the nodes are laid out on a line, it can estimate that the distance from A and this secondary nodes is equal to the distance from A to B, minus that from B to A.

If also long paired reads are used, as Velvet can accept mixed read lengths, another algorithm called Rock Band is used to build scaffold. Long reads can be used to connect nodes that are produced after the error correction. This algorithm is based on the idea that if a long read goes out of a unique node and leads to another unique node, these two nodes can be merged. As Pebble, Rock Band examines all the unique nodes starting from the one with bigger coverage towards to the one with less. For every unique node it counts the long reads that are going out from this node and if all reads go to the same destination the two unique nodes are merges and the gaps are filled with the long read information.

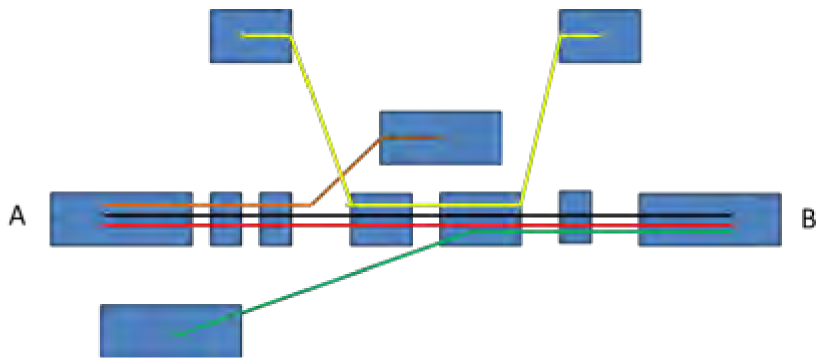


Figure 14

An schematic example that shows what Rock Band does. Contigs are the displayed rectangular and the reads are the colorful lines that pass through the contigs. Let's assume that the are two unique nodes A and B. The algorithm starts with examining the long reads going out from A. Two long reads, black and red, goes to the unique node B. The brown long read goes to a non-unique node so is discarded. Then all long reads going into node B are examined. All come from A except the green one that is discarded as it come from a non-unique node. In the end yellow long read that overlaps with the others reads is discarded as do not end in one of the unique nodes.

3.3 ABySS Assembler Description

3.3.1 Introduction

ABySS is a parallel sequence assembler. It uses a distributed de Bruijn graph in order to parallelize the assembly of a huge amount of small reads over a cluster machine. This implementation give us the opportunity to increase the memory that is available to the assembly process leading to an increase of the genome size that can be assembled. Summarizing we can say that that the main strengths of ABySS are the small memory foot print, the distributed processing with MPI and that can handle very large genomes.

AbySS is implemented in C++ and uses Message Passing Interface (MPI) for communication between the nodes of the cluster. The cluster's bandwidth and latency affect significantly the performance of the parallel assembly procedure. In order to limit the latency in each communication message corresponds a unique ID. The process sending the message does not wait for a respond immediately but is the current state of the operation using the ID and continues its execution processing other operations. The saved information is recovered when a response for a message is received with the ID of this message. Then the original task can continue its job. Moreover, in order to hide the latency of the network link the system allows many operations that can run in the same moment on each of the cluster used by AbySS.

3.3.2 Distributed De Bruijn Graph

The distributed de Bruijn graph is an implementation of the normal de Bruijn graph that allows storing neighboring sequences on the same computer. With this implementation the sequences are stored in different nodes of a cluster. In order to achieve this two things are needed. The one is to efficiently compute the the location of a k-mer from its own sequence, and the second is to store the neighboring information in a place that is independent of the location of the k-mer.

The location that a k-mer will be stored is computed through a hashing procedure. A value form zero to three is assigned to every base (i.e zero to Adenine, one to Cytosine, two to Guanine and 3 to Thymine) in order to form the representation of a k-mer. From this representation a hash value is computed. The same process is applied to the reverse complement of the k-mer and the 2 hash values are combined with the XOR operation. The outcome value from the XOR operation modulus the number of nodes provide the index that will determine in which node is going to be stored the k-mer.

The neighboring information is stored in a 8 bit per k-mer vertex. Since the alphabet has four symbols (i.e A, C, G and T) the maximum in-degree and out-degree of each vertex or k-mer is four. The presence or the absence of each edge is stored in a single bit. A value of 1 corresponds in the presence of an edge and 0 to the absence. Neighboring k-mers are generated from the information that the 8bit vertex provides and their location is computed with the hashing method described in the previous paragraph.

3.3.3 ABySS-Assembly Procedure

The main ABySS assembly process is done in three steps depending on the desirable assemble. The first is done without the use of pair-end information. Contigs are merged and extended until they cannot be extended any more due to lack of coverage or because a further extension will lead to unambiguity. The second step uses paired-end information to remove errors and extended more the contigs. And the third step uses mate pair information to build the scaffolds

De Bruijn Graph construction in ABySS

At the beginning of this step sequences with bases determined by N or "." are discarded and the data are loaded in the distributed de Bruijn graph. The k-mers are computed by shifting a piece of length k through the input sequence. So if the input sequence is length l, (l – k + 1) overlapping k-mers are created. Then with use of the hashing fuction described above, the cluster node index of every k-mer is computed and the k-mer is stored in the node that belongs to. A sequence is not stored in the hash table if the complement of this sequence is already registered in the hash table as they are considered equivalent. At the end, as all k-mers are loaded in the graph, start the computation of the adjancency. Every k-mer sends a message to the eight possible neighbors. If one exist must overlap in k-1 bases and this neighboring information is stored.

Error correction

In this step the algorithm cleans the graph from sequencing errors. The most common sequencing errors are the spurs. They are short dead end branches of the main path. As far as these sequences don't have an extension and they are usually unique they are located, traced backward until the deviation point and if the branch length is shorter than the threshold that was set they are eliminated. This threshold can if is not override by the user is automatically set by the algorithm. This procedure is done a lot of times and every time with a bigger threshold length to remove bigger branches that weren't eliminated in the previous iteration. This procedure is affected by the choice of k-mer length. A big k-mer length will lead to a big amount of short dead end branches and this will make difficult the distinction between a sequence error and a correct sequence that cause a branch due to lack of coverage. The latter case can results in contigs of a smaller length so the choice of the k-mer should be done carefully.

Another common sequencing error, caused in the middle of read, which can be located in a graph is a deviation of the main path in two branches that after rejoin together is called bubble. To remove bubbles ABySS locates every deviation point in the graph. Then each path from a deviation point is traced forward searcing for other paths that join after n nodes where is limited in the interval [k, 2k]. If a path that joins is found then the path with the lower coverage is removed and stored in a log file. Although, not only sequencing errors led to the formation of bubbles but also repetitive genome regions. In such case the removal of a bubble will minimize the repeat to a single sequence.

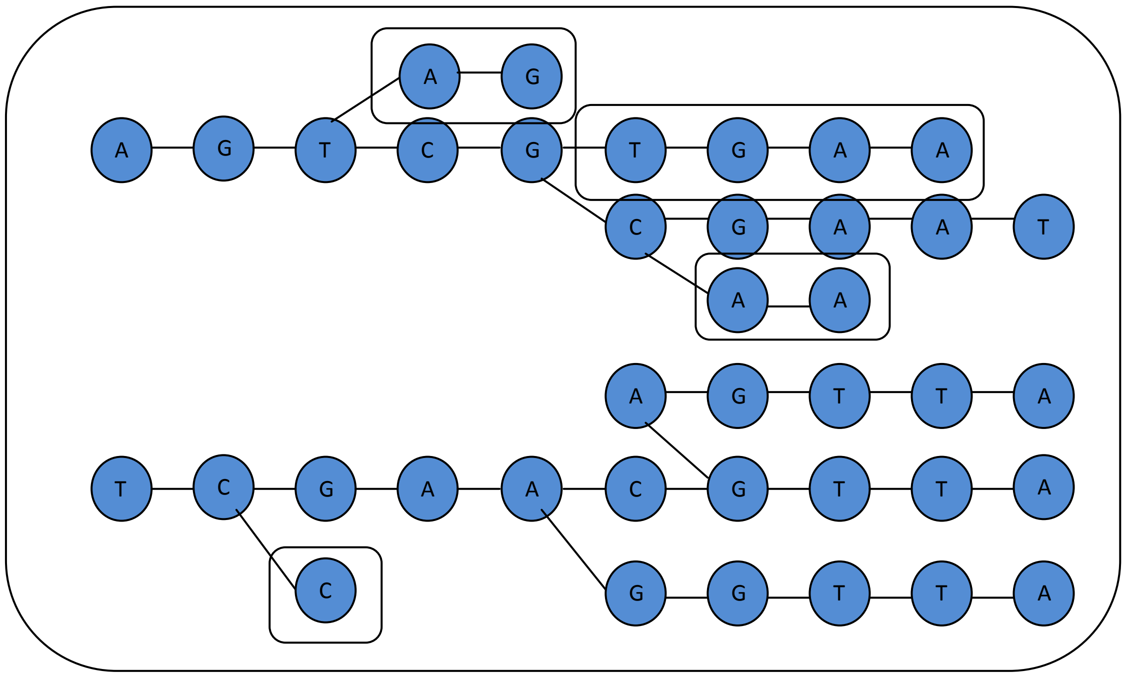


Figure 15
Short dead end branches are located. The branches iside the elipses are considered sequencing errors. These errors can be of length k-1 or less. The assembly procedure trim these branches to prevent a premature end of the algorithm.

In the next page:

Figure 16

Two bubbles: The first is a simple bubble that created from the deviation of two branches. The second is more complex as it consists in the intersection of two bubbles. The bubbles can be of length 2k-1 or less

Figure 17

The removal step: The first bubble is removed with the help of coverage. The path with the lower coverage is removed and saved in the Log file. The removal of the second bubble creates a dead end branch but I can also create a bubble of a lower level.plex as it consists in the intersection of two bubbles. The bubbles can be of length 2k-1 or less.

Figure 16

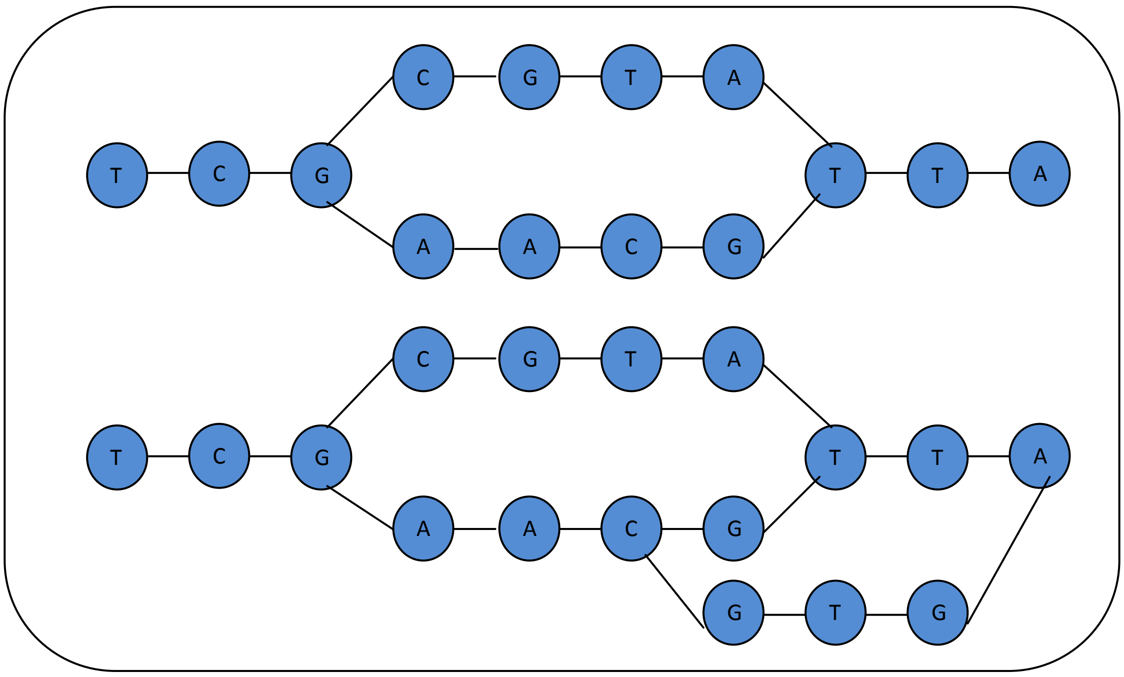
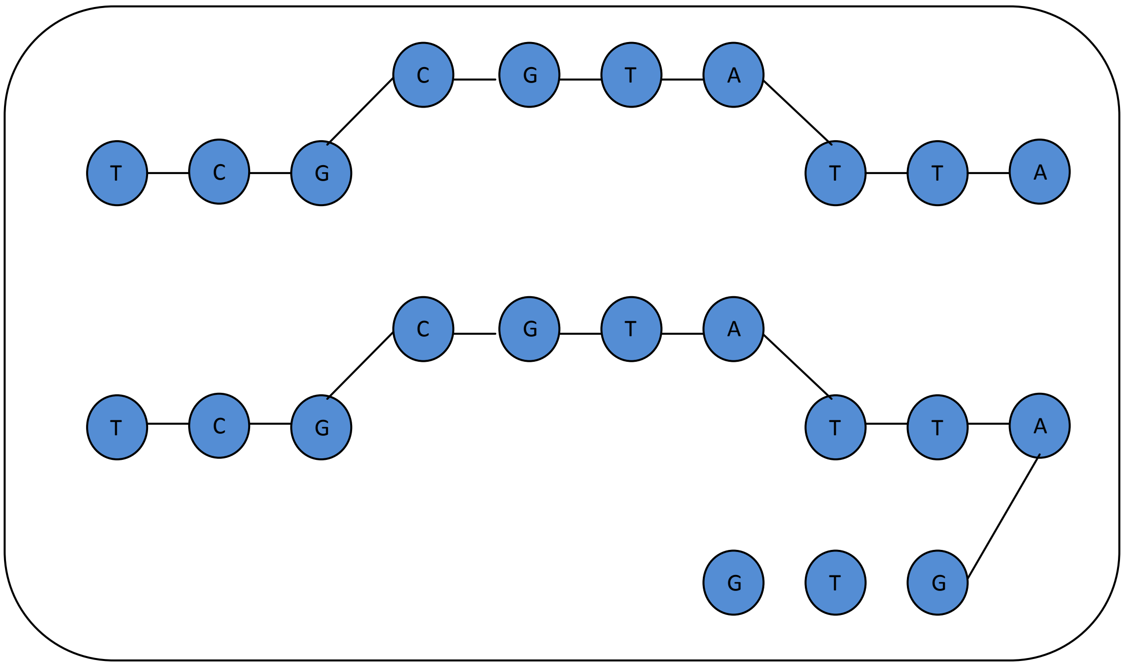

C G T A
T C G T T A
A A C G
C G T A
T C G T T A
A A C G
G T G

Figure 17


C G T A
T C G T T A
C G T A
T C G T T A
G T G

The removal step is shown. The first bubble is removed with the help of coverage. The path with the lower coverage is removed and save in the Log file. The removal of the second bubble creates a dead end branch but also creates a bubble of a lower level.

Merging Vertices

In this last step of the first phase the algorithm merges vertices that are connected via unambiguous edges. If there is an unsolved ambiguity in the in contigs extension the procedure of contig's length increase is stopped. Then the remaining connected nodes are merged creating independent contigs that overlap by no more than k-1 bases. The output contigs are in FASTA format.

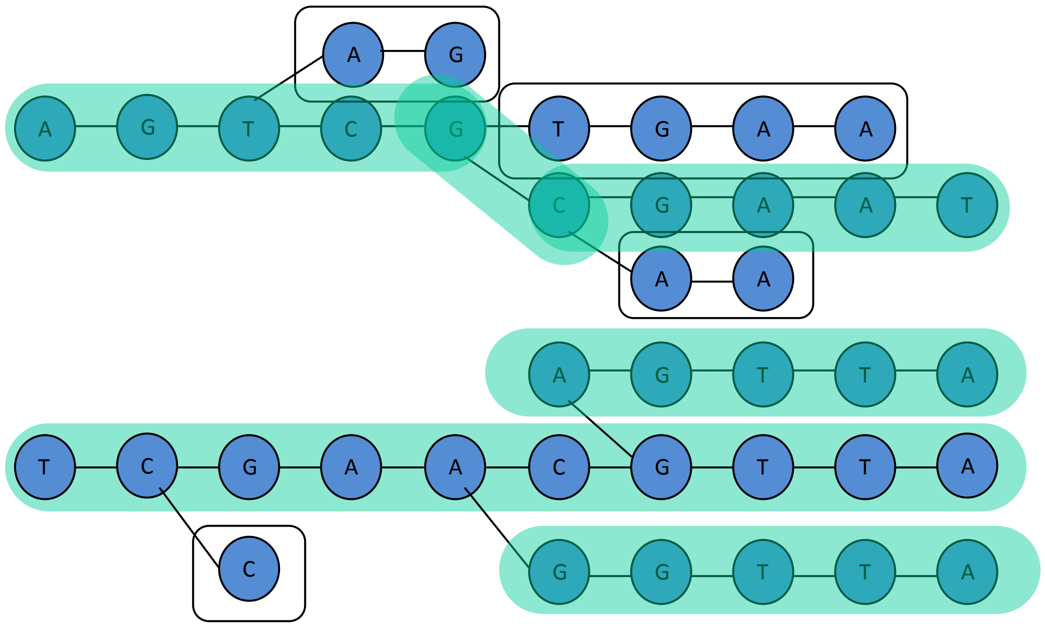


Figure 18

Unresolved ambiguities lead to the creation of lot of contigs. These are merged to create independent contigs in FASTA format

Second Phase-Use of pair-end information in order to merge contigs

In this phase the pair end information, if provided, is used to merge the previously created contigs. This information helps in the way of finding contigs that can be connected together by removing ambiguities. The contigs created in the previous phase are aligned, linked and then filtered to remove connecting errors between them. In detail every k-mer in the single-end assembly is considered to be unique. Then the algorithm maps reads with k consecutive correct bases. ABySS disposes a set of aligners that can be used in different cases. Some short and long read aligners are presented in the following table.

Short read Aligners

Aligner’s Name	**Strengths**
Bowtie	Fast
BWA	Useful with small gaps
GSNAP	Useful with big gaps

Long read Aligners

Aligner’s Name	**Strengths**
BLAST	Many reference genomes
BLAT	Useful with large gaps
BWA-SW	Useful with small gaps
Exonerate	Easy to use
GMAP	Useful with large gaps
MUMmer	Align two different genomes

Two contigs can be merged if at least p pairs join the contigs. For each contig, Ci, created in the previous phase, is generated a set of contigs, Pi, that are paired to this contig. Then a search in the de Bruijn graph is made to find a unique sequence of contigs from Ci that visits each Pi. Heuristics are used to limit the numbers of the visited vertices because the repetitive areas of the graph can lead to a huge computational cost. This procedure is repeated for all the contigs and at the end the paths are linked together to create the final contigs.

Third Phase- Use of mare pair information in order to build scaffolds

The basic idea behind scaffolding is that distance estimates are found in the same way paired end distance estimates are found, but the estimates aren't used so much for the distances as for linking information. A scaffold graph is formed from the distance estimates, and a number of transformations and heuristics are used to simplify the graph as much as possible. Then scaffolds are made along unambiguous paths where the number of N's inserted between contigs is related to the estimated distance between the contigs.

4 Magerit & SLURM Description

4.1 Introduction

In this chapter will be described the environment in which the tests were made. The architecture of Magerit and SLURM, that is the queuing system of Magerit, will be described briefly and a short description of how to submit a job in this cluster machine will be presented.

Magerit is the name of the one of the most powerful supercomputers of Spain. The second version, installed in 2011 reached the 1st position of Spain, 44th of Europe and 136th fastest of the world. This computer is installed in CeSViMa, a research center of the Technical University of Madrid. The experiments done in this study are made in this second version to take advantage of all the features that such a machine can give.

4.2 Magerit 2

Magerit is a cluster consisting of 260 nodes. The majority of them (i.e 245 nodes) are eServer BladeCenter PS702 2S with 16 cores in two 64-bit POWER7 processors (eight core each) of 3’3GHz and 32GB Ram. The rest 15 nodes are eServer BladeCenter HS22 with eight Intel Xeon 2’5GHz processors with 96GB RAM. The total system implies 4,160 CPUs and 9.2 TB RAM. All the nodes operate independently and all with the same software configuration. The system has a distributed storage system with a capacity of 192, TB provided by 256 disks of 750 GB each, which used a distributed and fault tolerant system (GPFS).

Moreover Magerit, due to its dimensions, process batch jobs with large processing requirements. In order to handle these jobs, which run in hundreds of CPUs a few days, it organizes them with a queue manager called SLURM as it is impossible to use more conventional access to the resources. SLURM plans the distinct jobs having as an object to maximize the use and the power of the computer and process user’s jobs as fast as possible without create starvation problems.

4.3 Simple Linux Utility for Resource Management (SLURM)

SLURM is an open source, fault tolerant system for high scalable cluster management and job scheduling used in Linux both small and large Linux clusters. This system provides the basic functions. First, it allocates exclusive and/or non-exclusive access to the nodes of the computer to the users for a duration of time giving them the chance to execute a job. Second, for every job, that is usually a parallel one, distributed between the available nodes, it provides a framework for starting, executing and monitoring this job. Third, it manages pending jobs and their requirements, which can create conflicts, with the use of a queue.

A queuing systems aims to provide a fairshare scheduling. This means that all users are tried to be served in a fairly way when they need resources. In detail, a job starts its execution before another one is based on two parameters. First when this job was submitted and second how many resources are available at this times to the users. Users that used less CPU time in the last job submissions has a priority from the recent more active users. Of course this convention takes place if resources are not available for all the user that submit a job at a certain time.

The architecture of slurm is based on a centralized manager that controls both the resources (i.e nodes) and the jobs by allocating them in the computer nodes. This manager called slurmctld implements also a management daemon. Each node also implements a daemon called slurmd that controls the tasks that are going to be executed in this node. It waits for a job, executes that job, return a status and waits for more jobs. Moreover it provides fault-tolerant hierarchical communications. Slurm contains also other daemons that are not going to be mentioned and analyzed. Other daemons are not going to be explained as the are not so relevant in this study. More information about slurm can be found here:[18]

SLURM daemons manage nodes, partitions, jobs and jobs steps. Partitions are sets of nodes collected in logical groups. They can be seen like job queues, each of which has a collection of constraints such the size limit a job, the time limit of a job, permissions that a user has in this partition etc. A job is a resource allocation for a specified amount of time. Jobs are allocated nodes within a partition until available resources, such as memory, nodes, processors etc. are exhausted. Finally, job steps are, usually parallel, tasks inside a job. Once a job is assigned in a set of nodes a user can start multiple job step in the allocation. Multiple job steps can run in an independent part of the allocation or a single job step can run in all the nodes. A schematic representation follows:

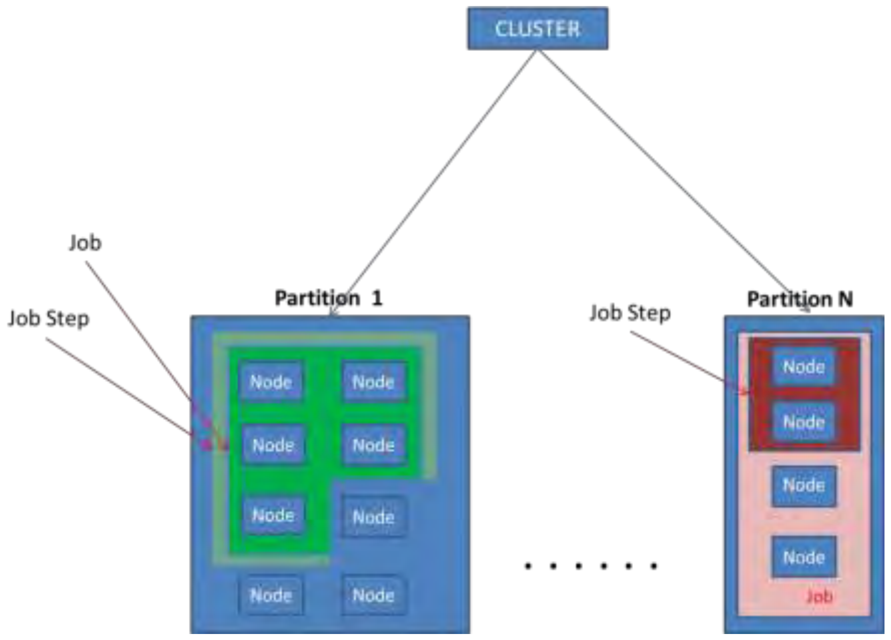


Figure 19

SLURM entities described above

4.4 Executing Jobs in Magerit

A job can be submitted by a user in a script format called batch or can be interactive and executed in real time. A user can add constraints to the execution, like for example how many nodes and how many processors this job is allowed to use, what of partition is going to be used for the execution, the time limit of the job etc. When the job is submitted the queue manager will try to find the resources defined in the job file among the available ones, optimizing the machine use and decreasing the waiting time of other users. In other words the queue manager tries to maximize the efficiency of the supercomputer.

A summary of the steps that the user should do in order to execute a job in Magerit is the following:

1. Connect to an interactive node
2. Prepare the executable that want to send in the supercomputer
3. Prepare the job definition
4. Send the job to the queue manager
5. Wait until the system assigns the nodes and executes the load of the job
6. Retrieve the results and send new jobs

Although some of the parameters that can be defined in the job file cannot exceed a predefined value. In Magerit, for example, the Quality of Service (QoS) implies an upper limit to the number of CPUs and the wall clock limit that each user has. A standard quality of service has the following characteristics. Every user, depending of the project that he is working for, can use a subset that will be assigned to him. In this way every time a user submits a job can indicate a different QoS from the ones that he is allowed to use.

Quality of Service	Cpus	Time
debug	16	00:10:00
class_a	1024	72:00:00
class_b	1024	36:00:00
class_c	1024	24:00:00
standard	512	72:00:00
premium	1024	72:00:00

A user not only can submit a job but also check the state of a job, block a job, see a list of the running jobs and cancel a job. These commands are available thought a SLURM-Moab interface that provides the following commands:

jobsubmit: Submit a job

jobcheck: Check the state of a job

jobhold: Block/unblock a job

jobq: List of running jobs

jobcancel: Cancel a job

As mention,in order to submit a job a batch file should be created. A typical batch file is presented in the next picture. It contains only the necessary declarations. These declarations and some optinial ones are described here:

#!/bin/bash

Indicates where shell is located in the system. By default all systems that have Bash installed they have shell under /bin directory. By using this line the script can be executed as a normal program.

#@ class

Indicates the QoS

#@ initialdir

Sets the working directory of the script. All the specified routes (output, error..) are relative to this directory. If not specified is considered to be the current working directory (.)

#@ output

Indicates where the standard output of the job will be redirected. This file contains the combined output of all the processes that took place during the execution of the job, directly or indirectly, in any of the assigned nodes.

#@ error

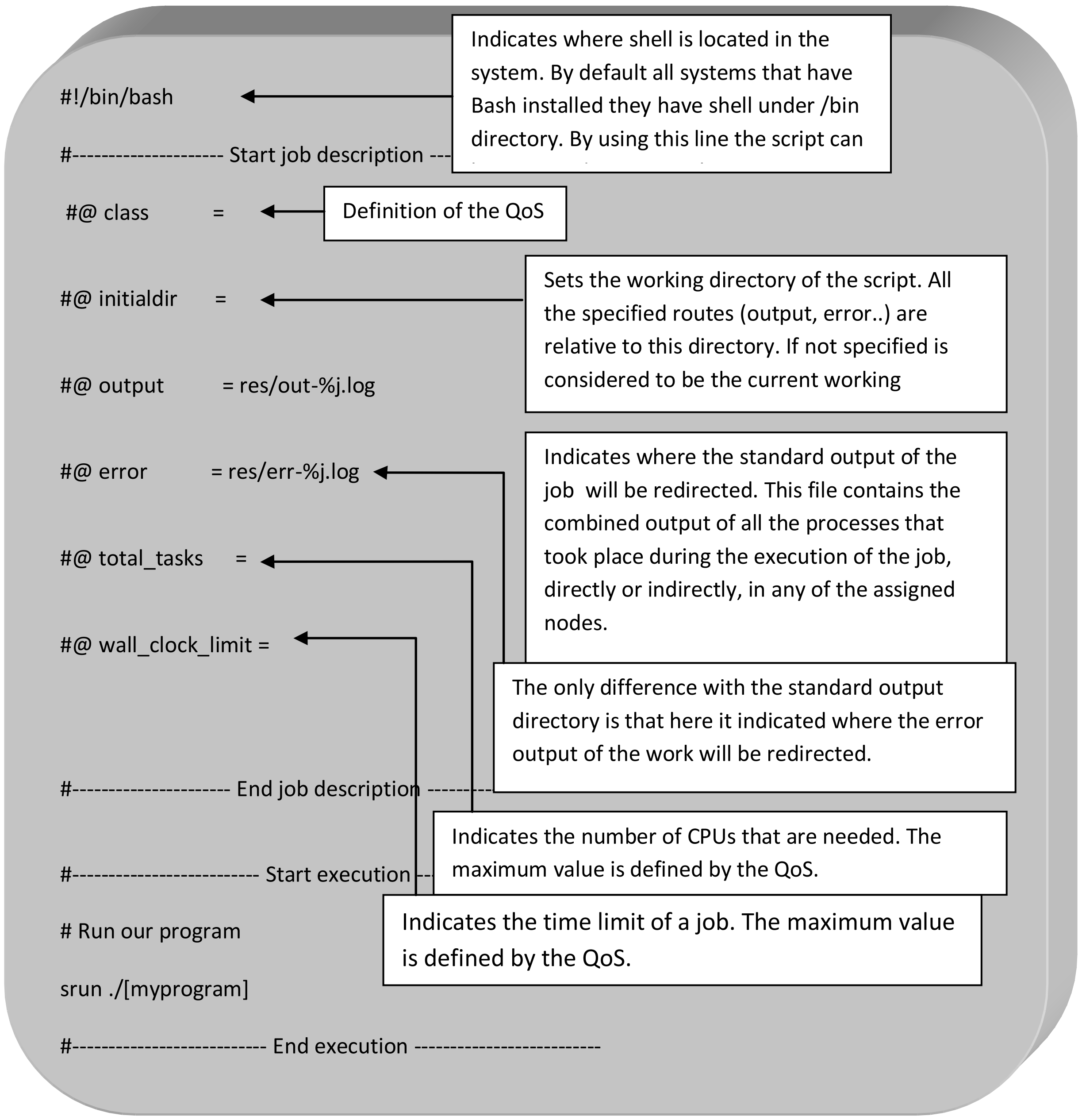
The only difference with the standard output directory is that here it indicated where the error output of the work will be redirected.

#@ total_tasks

Indicates the number of CPUs that are needed. The maximum value is defined by the QoS.

#@ wall_clock_limit

Indicates the time limit of a job. The maximum value is defined by the QoS



```
#!/bin/bash

#-------------------- Start job description ---

 #@ class         =

#@ initialdir     =

#@ output         = res/out-%j.log

#@ error          = res/err-%j.log

#@ total_tasks    =

#@ wall_clock_limit =

#--------------------- End job description --------

#------------------------- Start execution --

# Run our program

srun ./[myprogram]

#-------------------------- End execution ------------------------
```

Indicates where shell is located in the system. By default all systems that have Bash installed they have shell under /bin directory. By using this line the script can

Definition of the QoS

Sets the working directory of the script. All the specified routes (output, error..) are relative to this directory. If not specified is considered to be the current working

Indicates where the standard output of the job will be redirected. This file contains the combined output of all the processes that took place during the execution of the job, directly or indirectly, in any of the assigned nodes.

The only difference with the standard output directory is that here it indicated where the error output of the work will be redirected.

Indicates the number of CPUs that are needed. The maximum value is defined by the QoS.

Indicates the time limit of a job. The maximum value is defined by the QoS.

Figure 20

Presentation of a typical job with the most important parameters

5 Installation In Magerit

5.1 Velvet Installation

5.1.1Requirements

Velvet can be installed in a 32bit Linux environment with at least 12GB of physical memory. Although 32bit systems have memory limitations that can lead to restriction for the assembly process. To avoid these constraints Velvet should be installed in a standard 64bit Linux environment.

In the present study, Velvet version 1.2.03 was built in the 15 eServer BladeCenter HS22 nodes that have eight Intel Xeon 2.5GHz processors with 96 GB RAM to exploit the amount of memory and avoid as much as possible the assembly constraints created by the lack of memory.

5.1.2Compilation

In a GNU environment in order to install Velvet it is need only to type:

make

Lot of settings can be used in order to serve our needs in the assembly procedure. We used some of them to achieve better results and test Velvet in different ways.

Velvet version 1.2.03 was built in Intel nodes of Magerit with the following command:

make "OMENMP=1" "CATEGORIES=4" "MAXKEMRLENGTH=61"

- *OPENMP=1* allow to turn on multithreading. The program can use multiple cpus that are located in the same machine. In our case the number of CPUs can vary from 1 to 8. This option will lead to faster results especially when big genomes are used. To enable this option when an execution is made it is needed to set the environment variable OMP_NUM_THREADS=#CPUs in the job file that will be submitted. Velvet will use OMP_NUM_THREADS+1 CPUs to run the assembly procedure.
- *CATEGORIES=#* allows to distinguish reads from different insert libraries. By default Velvet uses on two short reads categories. This variable was extended to 4 as is the biggest number of insert libraries that is going to be used in this study. The bigger the number used in this variable the more memory will be required to run Velvet.
- MAXKEMRLENGTH=# allows to increase the hash length. The default maximum value used in Velvet is 31bp. The k-mer length determines in a big level the quality of the assembly. In this study Velvet was built with MAXKEMERLENGTH=60 in order to be tested with k-mers up to 60bp. Setting a bigger number for the hash length will require more memory in order to store longer words.

Other setting not used in the present study but that can be usefull are:

BIGASSEMBLY=1 allows to store more reads. If the libraries used contain more 2.2 billion reads Velvet needs more memory to store them. Setting BIGASSEMBLY=1 more memory is assigned in order to store these reads.

LONGSEQUENCES=1 allows to increase the read lengths that can be stored. By default read lengths are stored on 16bit signed integers. Because of this if longer than 32kbp reads are used in the process Velvet should build using LONGSEQUENCES=1 in order to provide the memory that is required to store coordinates.

5.2 Fatsx Toolkit

5.2.1 Introduction-Description

This toolkit was installed as Velvet expects that pair-end reads come from opposite strands facing each other. As in this study pair-end reads produced from the same strand are used, a reverse complement of these reads must be produced before running Velvet.

FASTX-Toolkit is a collection of tools that can process FASTA/FASTQ short reads. The available tools are described in the web site :

http://hannonlab.cshl.edu/fastx_toolkit/

From the available tools that FASTX-Toolkit provides in the present study only FASTQ/FASTA Reverse Complement was used, that produces a reverse complement of each sequence in a FASTQ/FASTA file.

5.2.2 Installation

In order to install FASTX-Toolkit in Magerit libgtextutils-0.6 or above is required. Both FASTX-Toolkit and libgtextutils can be downloaded from FASTX web site:

http://hannonlab.cshl.edu/fastx_toolkit/download.html

Libgtextutils-0.6.1 was downloaded and installed in the local path as follows:

./configure --prefix=/usr/local/libgtextutils

make

make install

After libgtextutils installation FAST-Toolkit - 0.0.13.2 was downloaded and installed. Before running FASTX's "configure" script, the environment variable PKG_CONFIG_PATH had to be set. The actions done were:

export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig:$PKG_CONFIG_PATH

./configure --prefix=/usr/local/Fastx

make

make install

5.3 ABySS installation

5.3.1 Requierements

AByss in order to be installed requires the following libraries:

[Boost](http://www.boost.org)

[sparsehash](http://code.google.com/p/sparsehash)

[Open MPI](http://www.open-mpi.org)

It requires also a C++ compiler that supports OpenMP.

5.3.2 Compilation

As the pair-end mode of ABYSS does not support SLURM, some addition and changes were made to abyss-pe script for adding this support. The code added to the script in order to integrate abyss-pe with SLURM was:

```
#Integrate with SLURM

ifdef SLURM_JOB_NAME
name?=$(SLURM_JOB_NAME)
endif
ifdef SLURM_JOBID
k?=$(SLURM_JOBID)
endif
ifdef SLURM_NTASKS
np?=$(SLURM_NTASKS)
endif
```

Moreover, the np option of abyss-pe specifies the number of processes to use for the parallel MPI job. Without any MPI configuration, this will allow you to use multiple cores on a single machine. To use multiple nodes for the assembly ABySS should be run we the use of the parameter np=(the number of threads that are going to be generated). The abyss-pe driver script will start MPI process like so: mpirun –np 8 ABYSS-P. Although, users in Magerit does not have the permission to run mpirun. Users can use only srun to submit their jobs. As in the Magerit scheme, srun call mpirun internally changes are needed in order to deal with this scheme. The changes to abyss-pe script are the following:

```
ifdef np
#   ORIGINAL: $(mpirun) -np $(np) ABYSS-P $(abyssopt)
$(ABYSS_OPTIONS) -o $@ $(in) $(se)
   srun ABYSS-P $(abyssopt) $(ABYSS_OPTIONS) -o $@ $(in) $(se)
else
   ABYSS $(abyssopt) $(ABYSS_OPTIONS) -o $@ $(in) $(se)
endif
```

where #ORIGINAL and ABYSS-P is the parallel (MPI) de Bruijn graph assembler used to run ABySS in parallel. ABYSS-P its called automatically from abyss-pe.

As the above changes are made the installation can take place. We suppose an installation of ABySS in the local directory (i.e /usr/local). As ABySS will install with MPI support, OpenMPI executables must be in the local path. To place the executables in the local path the command is:

module load openmpi

Next, environment variables CC and CXX are needed to be set. CC environment variable defines what C Compiler is going to be used while specifies the C++ Compiler that is going to be used. These variables are set as follows:

export CC=mpicc
export CXX=mpixcc

where mpicc is compile and link MPI programs written in C while mpicxx compile and links MPI programs written in C++. They both provide the options and any special libraries that are needed to compile and link MPI programs.
As the above preliminary steps are made the process continue to the main building part. The steps made are the following:

./configure --prefix=/usr/local/ABySS

make

install

The default maximum k-mer size is 64 and may be decreased to reduce memory usage or increased at compile time. ABySS was installed keeping this default value. Although this value can be change in the configuration as follows:

./configure --enable-maxk= #

Where # is the desired value.

5.4 Running Velvet in Magerit

In this paragraph are going to be analyzed only the parameters of Velvet used in this study. There is big amount of other parameters that can be used to achieve the desirable results that are described in detail in the Velvet Manual [31]

First of all as we built Velvet activating multithreading we have to define the number of threads that are going to be spawned during the assembly procedure. As mentioned, the only thing needed is to define the number of threads by setting the environment variable OMP_NUM_THREADS equal to the desired thread quantity – 1. Although, something very important is that when variable OMP NUM THREADS is set to n, velveth uses n + 1 in the parallelized part, but the speed up corresponds to n. Thus when OMP NUM THREADS = 1 velveth uses two threads but the time is equal to the serial version. This is a very important in this study as the time of Velvet's experiments will be analyzed.

In the present study Velvet will only be tested with pair end FASTQ data. Although Velvet supports also other formats like FASTA, SAM, BAM, ELAND, GERALD, FASTA.GZ and FASTQ.GZ. In pair-end modem, Velvet, handles only interleaved or "shuffled" fasta and fastq files, where each pair is seen one after the other. The read indexed as 1 is paired with the read indexed as 2, 3 with 4 etc. Although, because most of the pair end libraries are divided in two different files a merge of these files is required. In order to merge in one file the forward and the reverse reads from these two different files, an initial step is needed before the main assembly procedure starts. Velvet provides a Perl script to perform this preliminary step in order to "shuffle" the two files of each library used in the assembly. This script can be executed in the following way:

shuffleSequences_fastq.pl library_1. fastq library_2.fasta output_library.fastq or

shuffleSequences_fasta.pl library_1. fasta library_2.fasta output_library.fastq depending on the file format.

After shuffling short jumping libraries usually are produced with and RF orientation must need to complement reverse. To do this the script fastx_reverse-complent must be applied to the circularized shuffled library. The way to do this is:

fastx_reverse-complement -i shuffledlibrary.fastq -o reversedshuffledlibrary.fastq

where –i is for input and –o is for output. The assumption used in this script is that the library that is going to be reversed is Illumina. Although in this study the libraries are shifted to have the Sanger placement type. This shift needs the addition of the parameter *–Q 33* in the execution of fastx reverse complement script before declaring the input library. Depending on the sequence technology use -*Q* parameter take different values. This reverse and complement action in the circularized libraries must be applied only to one strand, depending on the orientation, and then the library can be shuffled.

After receiving the shuffled and reversed libraries the procedure can proceed to the main assembly steps. The first step is to create the files that Velvet will use to build the graph. The program that does this job is velveth. Velveth reads the sequence files and produces a hashtable and the output files Sequences and Roadmaps, which were described in the description of Velvet's algorithm.

To do this work velveth needs as an input an output directory were the produced Sequences and Roadmaps file are going to be stored, the k-mer length, the file format of this files that are going to be assembled, the read category that these file belong and then name of the files. The syntax can be described as:

velveth   output_directory   k-mer_length   -file_format -read_category   filename

The read category can be one of the following:

short, shortPaired, long, longPaired

Instead of using a k-mer lots of k-mer's length can be tested by replacing k-mer_length as follows:

Velveth output_directory   Start, Step, End   ...rest of the files...

This example tests all k-mers from the starting k-mer length (Start) until the ending k-mer length (End) with a step of Step. The output directory produced each time will have the following name.

Ouput_directory_k-merlengt

This way avoids doing all the redundant computations.

The next step consists in the execution of the main part of Velvet's algorithm. Velvetg is part that the de Bruijn graph is built and manipulated to arrive at the final results of the assembly process. The syntax of velvetg is as follows:

velvetg   out_directory   ...other parameters...

where the output_directory should have the same name with the output directory of velveth, as velvetg reads the Sequence and Roadmaps file that the former produced and stored in this directory to build the de Bruijn graph.

As only pair end libraries will be assembled, two parameters must be specified to activate the pair-end mode in Velvet: the expected insert length and the expected short-read k-mer coverage. The expected insert length is the length of the sequenced fragments that produced the reads. If is not known can be replaced from an estimation. The expected k-mer coverage is a parameter that is used in Pebble to resolve repeats. This parameter can be set as auto, can be set approximately or can be calculated after one execution of Velvet with the use of estimate-exp_cov script provided by Velvet.

If the examined sample is believed uniform, expected k-mer coverage can be set auto. When this is done, the algorithm will compute a histogram of k-mer coverages, excluding extreme values, then estimates the median and set this value to the variable exp_cov. By setting this parameter as auto another parameter, coverage cutoff (cov_cutoff) used in removing erroneus connections step, will be also set automatically (unless if it override by the user). The value that this parameter will have will be equal to the half of exp_cov value.

For example with expected k-mer coverage auto and insert read's length 1000 the systax is as follows.

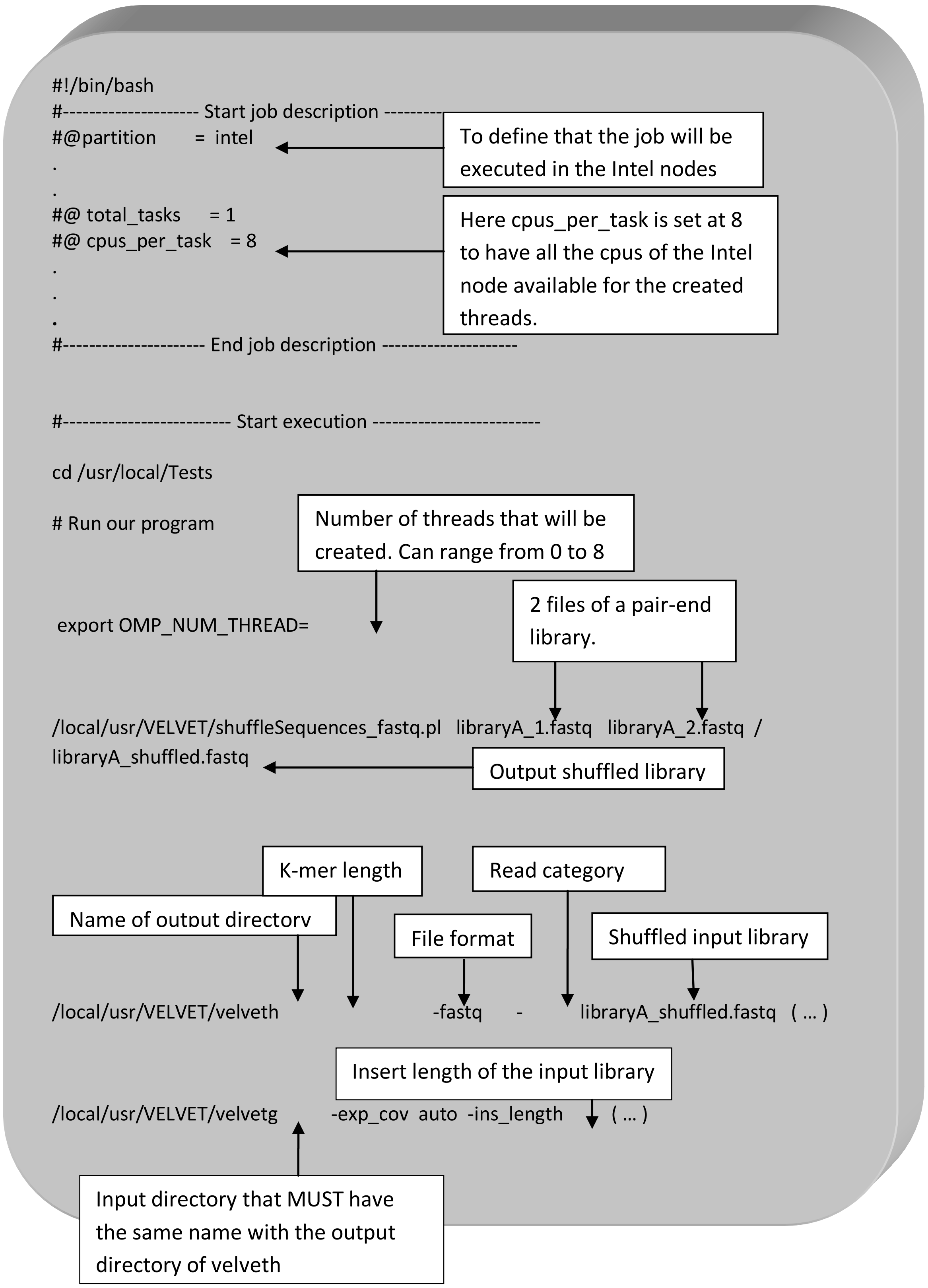
velvetg output_directory –exp_cov auto -ins_length 1000 ...other parameters....

An example of the batch file with the most used commands during Velvet pair end assembly is shown in the next page 57. Although more complex command can be used to achieve the desirable execution. In the job description part of the batch are mentioned only the parts that differ from the normal job description in the batch file. One fastq paired end library named A is used to explain the more important steps done.

If more libraries are going to be used for every library must be declared the format type, the read category and the insert length if is a pair-end (or a mate pair) library. For every additional library the read category and the insert length option must be followed by a number that show how many libraries are going to be used. To make this clear lets assume the use of 3 libraries all pair-end with instert lengths of 300, 500 and 1000. Then the execution of Velveth and velvetg will be:

velveth output_directory k-mer_length -file_format -shortPaired library1_name / –file_format –shortPaired2 library2_name –file_format –shortPaired3 library3_name / ..other parameters...

velvetg –exp_cov auto -ins_length 300 –ins2_length 500 –ins3_length 1000 / ..other parameters...


To define that the job will be executed in the Intel nodes
Here cpus_per_task is set at 8 to have all the cpus of the Intel node available for the created threads.
Number of threads that will be created. Can range from 0 to 8
2 files of a pair-end library.
Output shuffled library
K-mer length
Read category
Name of output directory
File format
Shuffled input library
Insert length of the input library
Input directory that MUST have the same name with the output directory of velveth

```
#!/bin/bash
#-------------------- Start job description ---------
#@partition      =  intel
.
.
#@ total_tasks     = 1
#@ cpus_per_task   = 8
.
.
.
#--------------------- End job description --------------------


#------------------------- Start execution -------------------------

cd /usr/local/Tests

# Run our program


 export OMP_NUM_THREAD=


/local/usr/VELVET/shuffleSequences_fastq.pl  libraryA_1.fastq  libraryA_2.fastq  /
libraryA_shuffled.fastq



/local/usr/VELVET/velveth                          -fastq     -      libraryA_shuffled.fastq   ( … )


/local/usr/VELVET/velvetg         -exp_cov  auto  -ins_length      ( … )
```

Figure 20

Presentation of a simple Velvet execution.

5.5 Running ABySSn in Magerit

Also in this paragraph are going to be mentioned the main parameters of ABySS and the one used in this study. More detailed explanation about every parameter that this assembelr can use can be found in ABySS Manual

As Velvet so ABySS is going to be tested with pair end FASTQ data. It supports also other formats like FASTA, SAM, BAM, QSEQ and SRA.The data can be comperessed with gz,bz2 or xz. Experiments of this study are going to be done with pair end data. In order to assemble pair end data with ABySS driver script abyss-pe need to be used. The suffix of the read identifier for a pair of reads must be one of '1' and '2', or 'A' and 'B', or 'F' and 'R', or 'F3' and 'R3', or 'forward' and 'reverse'. The reads may be interleaved in the same file or found in different files; however, interleaved mates will use less memory. In the experiments the suffix '1' and '2' will be used. abyss-pe has some input parameters that define what are the input libraries and how they are going to be used. In detail:

-in parameter is used to declare input files when assembling data from a single library

-lib parameter is used to declare a list of pair-end libraries when assembling data from multiple fragment libraries. For each library in lib, a variable with the same name must be declared specifying the files containg those reads

-pe is used to declare a list of paired-end libraries that will be used only in the step of merging contigs.

-p is used to declare a list of mate-pair libraries used for scaffolding.

-se is used to declare single end reads

To run ABySS in a pair end mode abyss-pe should be used defining the input libraries in one of the above categories. Moreover abyss-pe need some other parameters to start its execution. These parameter are the k-mer size andthe name of the output files. The can be declare as follows:

```
abyss-pe   name=output_name    k=k-mer length  ...other parameters....
```

This execution will produce all the assembly results in the path defined in the batch file. The destination of the results can be changed in order to store the results there. Also different k-mers can be tested to find an optimal value and their results stored every time in a different file. An example follows:

```
for k in {st_val..end_val}; do
        mkdir k$k
        abyss-pe   -C k$k   name=example     ...other parameters...
done
```

In order to limit the interval of the k-mer lengths that is good to examine in order to find which k-mer value leads to a decent assembly is reported that for ABySS :

"The *k*-mer value should not be less than ($l_{\max}$+1)/2, where l_{max} is the length of the longest read in the data set and cannot be more than the length of the shortest read, l_{min}. Although the theoretical lower limit for an assembly is *k*=2, the above bound is necessary to prevent excessive bubble formation, which otherwise would be possible for read errors that are observed only once. The upper limit is from a hard constraint, as for a *k* value higher than l_{min} it would not be possible to construct *k*-mers for reads of length l_{min} if *k* is beyond this limit.''

In order to run ABySS in a parallel mode abyss-pe must be used. Moreover, with the changes made in the abyss-pe script, the parallel execution will be enabled only when parameters total_tasks and cpus_per_task are set. abyss-pe script can be used for single-end, a pair-end assembly or an assembly that uses mate pairs. The mode that this script will run depends from the declared variables. Every of these parameters add the execution of an additional assembly stage:

Assembling without paired end information
abyss-pe unitigs …

Including the paired end assembly:
abyss-pe contigs …

Including scaffolding (use of mate pair information):
abyss-pe scaffolds …

ABySS produces a lots output temporary files that are needed between the assembly steps. These files for big assemblies can have a big size and Magerit the users space for temporary files is limited to 100MB. In order to change the location that the files are going to be stored the environment variable TMPDIR should be set with a location of a bigger size. In Magerit this location can be scratch file. The variable is set as follows:

export TMPDIR = /scratch

Another parameter very usefull parameter useful in order to see in detail every step of the procedure is *v*. This parameter also provide the memory consumption of the assembly that is need when a benchmarking is done. I order to enable verbose logging parameter

v=-v

must be added in the execution line of ABySS.

In figure in page 60 is shown a batch file with four diffents assembly executions in Magerit. All of them use abyss-pe to run the procedu re. As mentioned, more parameter can be added to the execution but this study concentrate in some of them.

The parameters in the batch file cpus_per_task and total total_tasks define in how many nodes and cpus ABySS is going to run. For example using:

total_task = 64

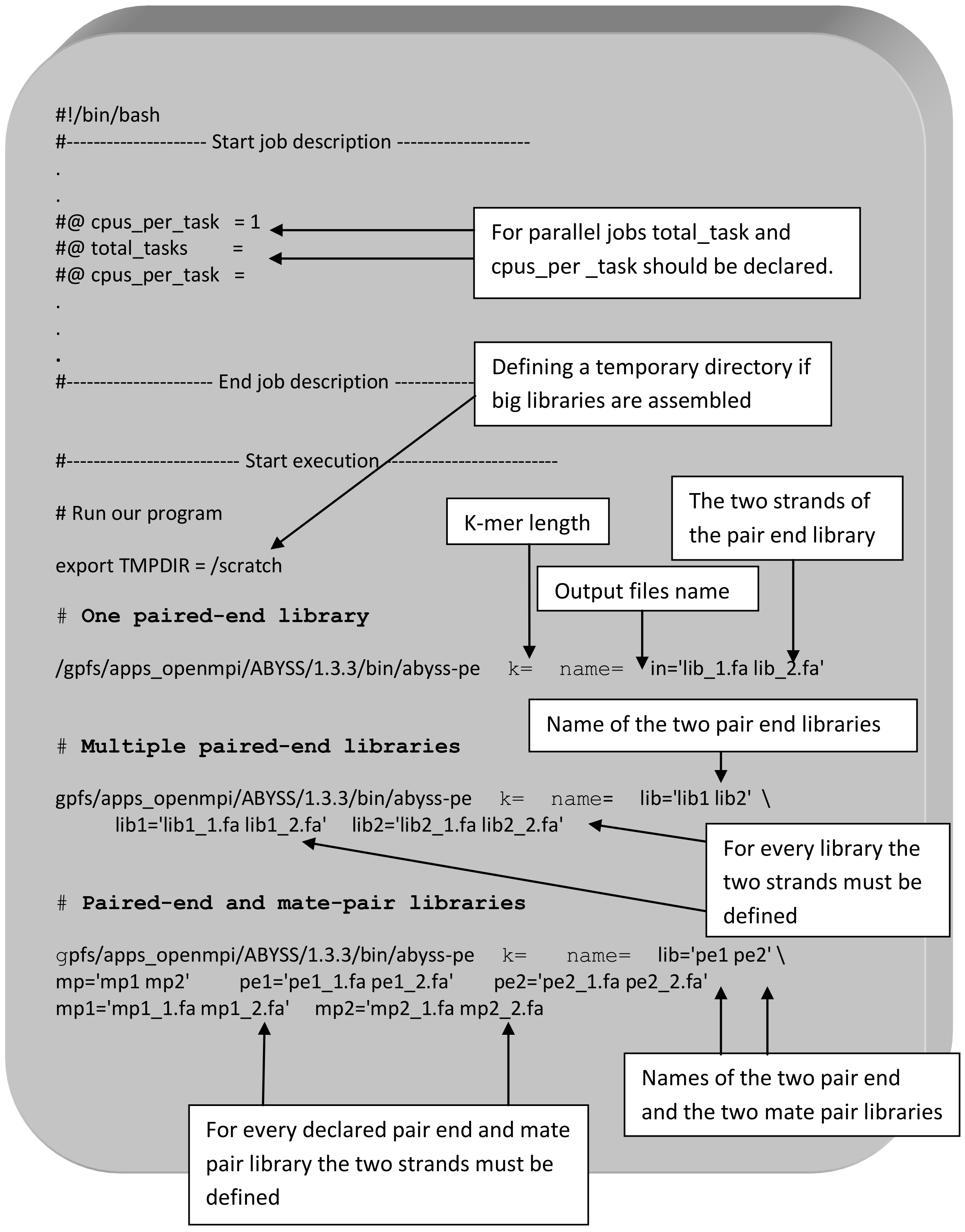
cpus_per_task=16

ABySS is going to be executed in 64 CPUs that belong to exactly 4 nodes. By defining cpus_per_task = 16 the user show his intention to use for his job a whole node alone so other jobs will not affect the running one. Thus in the above example 64 tasks will be distributed in 4 nodes running at full time as they will not be affected from other execution.

One last thing that must be mentioned is that in order to execute the latest version of ABySS the path is always:

/gpfs/apps_openmpi/ABYSS/1.3.3/bin/

followed by the desired script as ABySS is installed in the supercomputer.



```
#!/bin/bash
#-------------------- Start job description --------------------
.
.
#@ cpus_per_task  = 1
#@ total_tasks     =
#@ cpus_per_task  =
.
.
.
#--------------------- End job description -----------


#------------------------ Start execution ------------------------

# Run our program

export TMPDIR = /scratch

# One paired-end library

/gpfs/apps_openmpi/ABYSS/1.3.3/bin/abyss-pe   k=   name=  in='lib_1.fa lib_2.fa'


# Multiple paired-end libraries

gpfs/apps_openmpi/ABYSS/1.3.3/bin/abyss-pe   k=   name=   lib='lib1 lib2'  \
       lib1='lib1_1.fa lib1_2.fa'   lib2='lib2_1.fa lib2_2.fa'


# Paired-end and mate-pair libraries

gpfs/apps_openmpi/ABYSS/1.3.3/bin/abyss-pe   k=    name=   lib='pe1 pe2' \
mp='mp1 mp2'      pe1='pe1_1.fa pe1_2.fa'      pe2='pe2_1.fa pe2_2.fa'
mp1='mp1_1.fa mp1_2.fa'    mp2='mp2_1.fa mp2_2.fa
```

For parallel jobs total_task and cpus_per _task should be declared.
Defining a temporary directory if big libraries are assembled
K-mer length
The two strands of the pair end library
Output files name
Name of the two pair end libraries
For every library the two strands must be defined
Names of the two pair end and the two mate pair libraries
For every declared pair end and mate pair library the two strands must be defined

Figure 21

Presentation of four types of assemblies in a bath file mode.

6 Experiments

6.1 Introduction

The assemblies done in Magerit aimed in testing one shared memory assembler, i.e. Velvet, and one distributed memory assembler, i.e. ABySS, in order to check their performance in time and find out which assembler can give the best results in a shorter time.

The performance and the efficiency of an assembler are usually determined by the resource consumption and by the size of the contigs and scaffolds that that the assembly process produce. The resource consumption of an assembler consists in finding out the whole time of the assembly procedure and the RAM that is consumed. The measurement of the contigs and scaffolds size includes the definition of the N50, the maximum contig length and the total number of contigs that are produced.

Contig or scaffold N50 is a weighted median statistic such that the 50% of the entire assembly is contained in contigs or scaffolds equal or larger to this value. This statistic provides a way to measure the connectivity of the assembly where higher N50 lengths show better performance.

The most common way used from assemblers to define this parameter is described with the following steps:

1. All contigs are short by size
2. Contigs sizes are added, one by one, from the largest down to the shortest
3. When the size of the added contig's length covers the half of the genome this procedure is stopped.
4. The length of the last encountered contig is the N50 of the assembly

An example: Let's assume a genome of 40 Mbp. The contigs produced after the assembly processes are shown in the figure below. From the length of the contigs shown the N50 statistic will be set as 4.5 Mbp because 7+6.5+4.5+4.5=22.5>20

The N50 statistic has some characteristics that are very useful to determine the efficiency of the assembly. These characteristic are the following:

1. High N50 means long contigs and thus a good assembly
2. Low N50 means many short contigs, genome bad sequenced and thus a bad assembly

From the things mentioned above its clear the the N50 statistic is very useful to determine the quality of an assembly. Although bad quality assemblies can have a high N50 statistic.

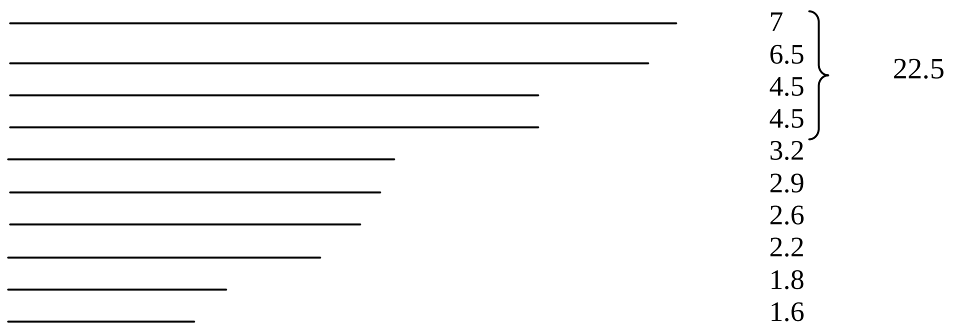


Figure 22

Contigs produced and their lengths in Mbp. The first 4 lengths contigs have length greater than the half of the assembled genome so the length of the fourth contig is the N50 statistic.

”The standard of judging assembly quality by size of contigs is questionable. Large contigs may simply reflect overly aggressive joining of contigs, thereby creating larger contigs with misassemblies. As a consequence, genome scientists who are not experts at assembly can be completely misled by statistics about contig sizes, and as a result might prefer the ’larger’ but incorrect assembly when given a choice.”[32]

The maximum contig length, also used to determine the quality of the assembly product, is an important parameter as longer contig mean that most of the errors were eliminated and the genome were assembled in fewer and longer contigs.

In the experiments done in Magerit were tested three genomes. These genomes belong to a Staphylococcus aureus, an Escherichia coli and a Rhodobacter sphaeroides. All the used fragment pair-end libraries, described in detail in the next paragraph, have reads of 101 bp that is a relative big value for the examined short read assemblers. The k-mer lengths used were odd and they belong to the interval [19,61]. The values where chosen to be odd as Velvet handles only odd k-mer lengths and the chosen interval reflects the need to take advantage of the provided coverage with respect to the specificity that relative long k-mers provide. Bigger k-mer length, that belong to the interval [60,80], will be also good to be checked but in this study we limited in the mentioned interval to check the performance with smaller k-mer lengths.

The tests that this study aimed to do with Velvet were:

For Staphylococcus aureus and Escherichia Coli to test the k-mer lengths 25, 35, 45, 55 and then depending on which k-mer length were observed the biggest N50 to test the nearby ±3 odd k-mers length to define the best k-mer length in the interval [19,61]. Then, using this k-mer, to use the multithreading ability of Velvet to see how the use of more threads affects the processing time of both velveth and velvetg.

For Rhodobacter sphaeroides to test the k-mer lengths 25, 35, 45, 55 and then depending on which k-mer length were observed the biggest N50 to test the multithreading ability of Velvet to check the OpenMP ability of Velvet with respect to the spawned threads.

While the tests aimed to do with ABySS were:

For Staphylococcus aureus and Escherichia Coli to test the k-mer lengths 25, 35, 45, 55 using the short jumping libraries as mate pair libraries. Then depending on which of the above tests where observed the biggest N50 to test the nearby ±3 odd k-mers length and, like in Velvet, define the best N50 in the interval [19,61]. Finally, using this k-mer, test the MPI ability of ABySS with a different number of CPUs every time to check how the processing time is affected.

For Rhodobacter sphaeroides to test the k-mer lengths 25, 35, 45, 55 with the provided jumping libraries as mate pair libraries in the scaffolding stage. Then, using this k-mer, to test the MPI ability of ABySS with a different number of CPUs every time to check again the affection of the processing time.

Also for the multithreading tests of Velvet and for the MPI tests of ABySS will be benchmarked the peak of the consumed memory in order to have a full view of the resource consumption during the mentioned tests.

**The commented results that are not present in the charts for every experiment can be found in the appendix*

6.2 Libraries

In this paragraph are presented the libraries used in order to make the experiments and test the two assemblers in an increasing order based on the size of these libraries.

Genome: Staphylococcus Aureus		
Genome's Size: 2.839.460 base pairs		
Library Type	Pair-end Fragment library	Short Jumping Library
Sample	SRS004752	SRS004751
Run	SRR022868	SRR022865
Library	Solexa-8293	Solexa-3932
Average Read Length	101bp	37bp
Insert Length	180bp	3500bp
Number of Reads	1,294,104	3,494,070
Read Orientation	Forward Reverse	Reverse Forward
Run Base Count	131 Mb	129 Mb
Instrument Model	Illumina Genome Analyzer II	Illumina Genome Analyzer II

Genome:Escherichia coli (1)		
Genome's Size: 4.639.675		
Library Type	Pair-end Fragment library	Short Jumping Library
Sample	SRS009994	SRS269404
Run	SRR034509	SRR401827
Library	Solexa-11748	Solexa-44956
Average Read Length	101bp	93bp
Insert Length	180bp	5000bp
Number of Reads	10,353,618	1,615,703
Read Orientation	Forward Reverse	Reverse Forward
Run Base Count	2 GB	300MB
Instrument Model	Illumina Genome Analyzer II	Illumina HiSeq 2000

Escherichia coli (2)				
Genome's Size: 4.639.675				
Library Type	Pair-end Fragment library		Short Jumping Library	
Sample	SRS302375		SRS269404	
Run	SRR447625	SRR447685	SRR401827	SRR492488
Library	Solexa-25396		Solexa-44956	Solexa-42866
Average Read Length	101bp		93bp	
Insert Length	180bp		5000bp	
Number of Reads	13,479,432	13,457,571	1,615,703	362,200
Read Orientation	Forward Reverse		Reverse Forward	
Run Base Count	2.8GB	2.8GB	313.4MB	67.4MB
Instrument Model	Illumina HiSeq 2000			

Rhodobacter sphaeroides			
Genome's Size: 4.607.000			
Library Type	Pair-end Fragment library	Short Jumping Library	
Sample	SRS004732	SRS004732	
Run	SRR125492	SRR034527	SRR034528
Library	Solexa-11749	Solexa-11767	
Average Read Length	101bp	101bp	
Insert Length	180bp	4000bp	
Number of Reads	11,339,101	17,746,938	20,162,859
Read Orientation	Forward Reverse	Reverse Forward	
Run Base Count	2.3G	3.6 GB	4.1G
Instrument Model	Illumina Genome Analyzer II		

6.3 Staphylococcus Aureus

Velvet:

As, mentioned before starting a pair end assembly Velvet needs to shuffle the pair end libraries (both the fragment and the jumping libraries) that are divided in two files, one with the forward and one with the reverse reads. Moreover, as the second library has an Reverse Forward (RF) orientation it is needed to reverse complement both strands before shuffling or to reverse complement the whole previously shuffled library. The second way was the one used. The time of each action done in this preliminary step is the following:

Action	Input Size	Time
Shuffle fragment libraries	2x140 MB	0:00:26
Shuffle short jumping libraries	2x 161.41 MB	0:00:38
Reverse complement shuffled short jumping libraries	322.82 MB	0:00:04

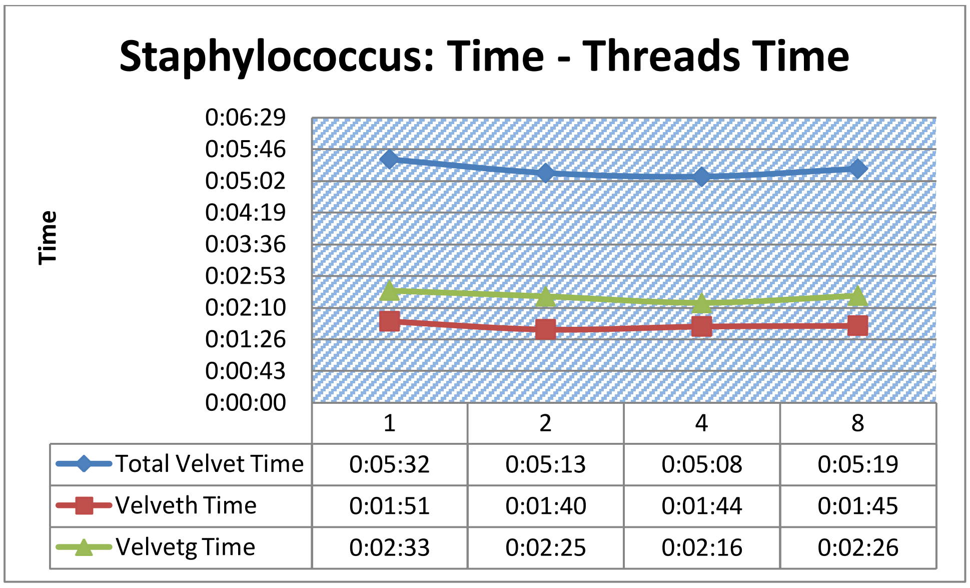
The total time of this step is 1.08 minutes and is a constant that will be add every time to find out the total time of the assembly.

After the above preliminary step the best k-mer in the interval [19,61] were defined in two steps. First the k-mer lengths 25, 35, 45, 55 where tested to define the biggest N50 value between them and then k-mers lengths around this k-mer value were tested to find out the biggest k-mer in all the interval [19,61]. The results of the steps are shown in Chart 9 and Chart 2**.** As shown the k-mer length that produced the better results is 35.

In the first test k-mer lengths 25, 45, 55 gave a very low N50 (especially k-mer length 55) output showing that they are unsuitable for testing this genome and. Moreover the use of a k-mer length produced a missassembly as the length of the output contigs is bigger than the genome size. A thing to be noticed, for the correct assembly, is that the scheme big N50 → few and long contigs and small N50 → lot and small contigs is verified

In the second test N50 statistic does not have a big variation in the interval [29, 37] while for the two last k-mer, 59 and 61, this statistic falls in indecent results. A very important point here is that the use of k-mer lengths 29, 31, 33, 37 and 39 led to missasemblies as the length of the output contigs is bigger from the escherichia coli's genome. An observation in this test is that with exactly the same parameters two runs with k-mer length of 35 had a N50 output that differs in ~60kbp. This deviation in the results is caused by the multithreading. The fact that the reads are not processed sequentially produces some discrepancies. This factor can lead sometimes in a "bad" choice as a better k-mer length can be discarded if in a given multithreading run has a smaller N50 from another k-mer length that is apparently more appropriate.

As the best N50 is defined the next step is to tested the OpenMP ability of Velvet. Results are shown in Chart 1 and Table 1



	1	2	4	8
◆ Total Velvet Time	0:05:32	0:05:13	0:05:08	0:05:19
■ Velveth Time	0:01:51	0:01:40	0:01:44	0:01:45
▲ Velvetg Time	0:02:33	0:02:25	0:02:16	0:02:26

Chart 1

For staphylococcus aureus: Time in minutes used by velveth, velvetg and the total time of the procedure as a function of the number of threads.

Percentage of time gained			
Cores	**velveth**	**velvetg**	**Total Time**
2	10.0%	5.2%	5.7%
4	6.2%	11.1%	7.2%
8	5.4%	4.5%	4.0%

Table 1

For staphylococcus aureus: Percentages of gained time for a single to a multithreaded execution

In detail we can see from both tables above that the whole procedure does not scale very well. With the use of four cores we have an decrease in time of ~7.2% that compared to the other values in not that big. Also velvetg with the use of four cores achieve his maximum speed up while velveth achieve this speedup with the use of two cores. The creation of threads includes an overhead, mainly caused by the creation of threads and the memory copied each time, that is compensate with the advantages that a multithreading execution provides. The use of eight cores means the creation of eight threads and more memory copies that from one hand make the move from single to multi-threading faster but compared with the use of two or four cores the overhead introduced is more visible and slows the whole procedure. This is mainly because the libraries used are of a small size and multhreading have some limits in the benefits that provides.

ABySS:

N50

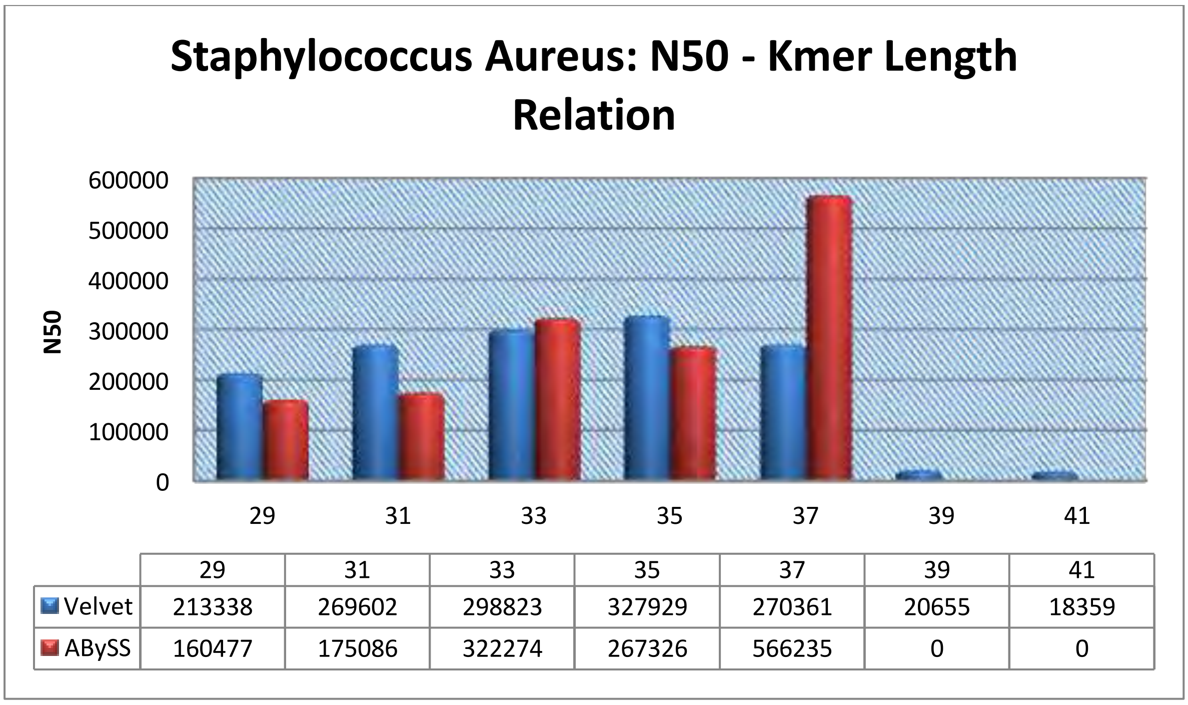
Same tests were made with ABySS to determine the best k-mer in the interval [19,61]. The first step is shown in Chart 9 and the second in Chart 2.

In the first test with k-mer lengths 25, 35, 45, 55 the first thing that someone can notice is that with k-mer lengths 45, 55 the output N50 is 0. This values does not reflect the reality, as the execution was stopped because with a k-mer above 37 is impossible to find alignments with the mate pair library as it has reads of 37bp. Moreover as k-mer length of 25 lead a missassembly the only value that we can choose, that also produced the biggest N50 value, is 35.

In the second test is more obvious that k-mer lengths above 37 are inappropriate as more values are checked. To overcome this result of an interrupted execution the parameter “l”, that is set automatically equal to the k-mer length, can be override by the user with a smaller value. This will lead to a complete assembly run but as in this study the default value of the alignment parameter “l” is used we assume that with k-mers bigger than the 37 the assembly procedure is stopped. In the interval of k-mer lengths [29,33] the N50 gradually increase but all the assemblies are indecent because they produce contings of a total length bigger than the genome size. The next two k-mer lengths, before the ones that lead to an interruption of the procedure, end in a correct assembly process from these two runs the one with k-mer length of 37 was chose due to the big N50 value.

With the chosen k-mer length we proceed to test the execution time of ABySS in different nodes. The results are shown in Chart 8.

Between the use of 1 and 2 nodes there is only a difference of a few second. When MPI is enabled, the use of two nodes seems to have the same behavior with the full use of 1 node. Things start to change with the use of three nodes. Time with the use of three nodes decreases of about ~30.5 % from the single node and of about ~25.4% when to nodes are used. The use of three nodes in the experiment seems to be the appropriate as with more nodes times starts to increase again and with the use of 112 and 128 CPUs ,i.e. 7 and 8 nodes respectively, the time becomes bigger from the one achieved with the use of only one node. With the use of 8 nodes the increment in time from the fastest execution is of about ~143.2% showing the MPI spends more time in communication than in the main procedure of assemble. This experiment shows that for small sizes of libraries the MPI ability of ABySS is limited to the use of a small amount of cores.



	29	31	33	35	37	39	41
🟦 Velvet	213338	269602	298823	327929	270361	20655	18359
🟥 ABySS	160477	175086	322274	267326	566235	0	0

Chart 2

Staphylococcus Aureus N50 values for both Velvet and ABySS in relation with the k-mer length

6. 4 Escherichia Coli (1)

Velvet:

Like in the first genome, also in this one the pair end and the short jumping library need to be shuffled. More over the short jumping library that has a Reverse Forward orientation need a reserve complement action. For this three preliminary steps the time was recorded as is a constant that every time need to be add in order to define the whole time for an assembly

Action	Input Size	Time
Shuffle fragment libraries	2x2.50 GB	00:07:35
Shuffle short jumping libraries	2x374.53 MB	00:01:02
Reverse coplement shuffled short jumping libraries	749.06 MB	00:01:05

The total time of this step is 9.42 minutes.

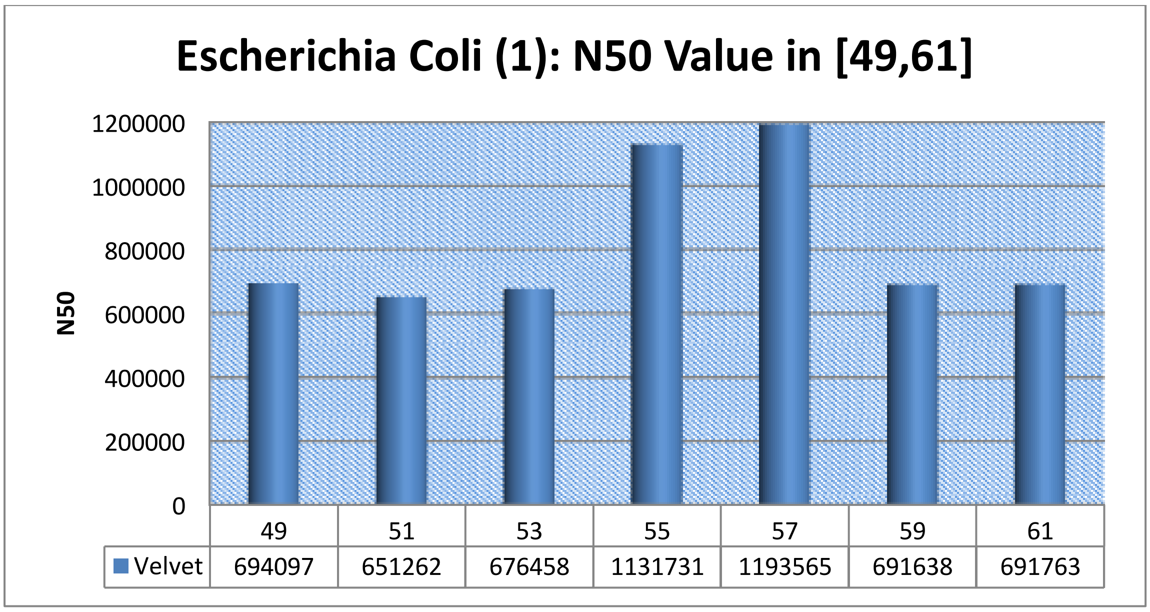
After the described above action the same test made to Stapylococcus Aureus were made to the Escherichia Coli genome shown in Chart 9 and Chart 3.

In Chart 9 it is obvious that with k-mer lengths of 25 and 35 Velvet N50 value is very small and the total length of the output contigs is ~18 times in the first test and ~4 times in the second more than the genome size. This results show that with k-mer lengths of 25 and 35 the procedure leads to non concrete results and these lengths should not be used. With the two next k-mer values the results are totally different. The N50 produced is more that 17kKbp times higher and the length of the total contigs is smaller from the examined genome size. The N50 value between k-mers 45 and 55 does not have a big variation and maybe the k-mer length of 45 produce better results if multithreading was not causing discrepancies. Although as we base our choice in the better actual N50 statistic the next step of finding the best k-mer is using as a reference the k-mer length of 55.

In the second test with all the ±3 around 55 the resulting N50 outputs have small differences between them with only two k-mers length procucing two N50 outputs of almost the double value of the others. This k-mers lengths are 55 and 57. In the previous test the N50 using a k-mer of length 55 was almost the half. The characteristic of multithreading to be non-deterministic causes big differences in the output values between same executions and this can lead to wrong decisions. An equilibration in these results can be the max contig length. Here the execution with a k-mer value of 51 that resulted in the smaller N50 has a bigger max contig value compared to executions with bigger N50. This means that the execution produced long output contigs and that these contigs where fragmented much more than with other executions leading to a bigger amount of final contigs. In the executions with a k-mer length of,55 and 57, the N50 has a value that is quite near to the longest contigs produced in this assemblies. This mean that with the use of these k-mer lengths the assembly was able to cover the half of the genome with the first few long contigs meaning that the quality of it was relatively good.

Choosing 57 as the best k-mer length we continued to the third test: to test the multithreading abity of Velvet. The results are shown below in Chart 4 and Table2. In the char we can see that the time decreases almost linearly. The use of more cores produces a decrement in time for velveth and velvetg leading in a fastest output of the whole procedure. Both velveth and velvetg give the output in almost the half time from the single threading execution with the use of eight cores. Especially velveth decreases the time spend to create the files in more than the half of the first execution. The overhead that the multithreading introduces its fully overcome the advantages of OpenMP are clearly visible in this test.

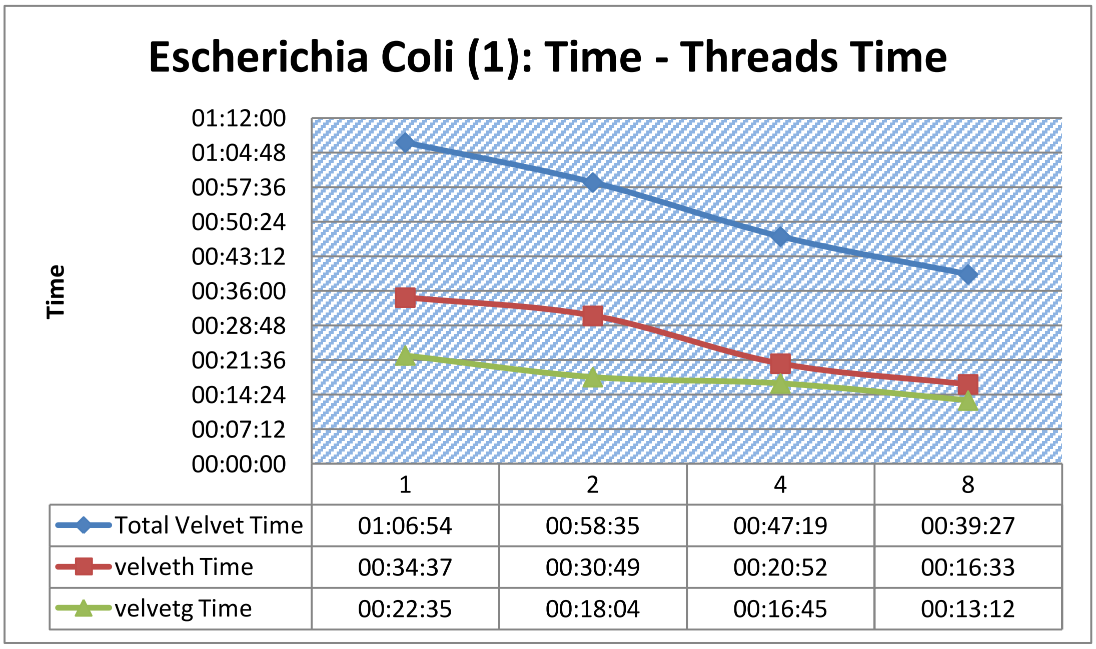
Something last visible in the results of this test is the N50 value. The differences in the N50 statistic between the same execution with different number of threads range from ~700bp to ~500000bp showing that this can be a drawback of a multithreading execution.



	49	51	53	55	57	59	61
🟦 Velvet	694097	651262	676458	1131731	1193565	691638	691763

Chart 3

Escherichia Coli (1) N50 values in interval [49,61]



	1	2	4	8
◆ Total Velvet Time	01:06:54	00:58:35	00:47:19	00:39:27
■ velveth Time	00:34:37	00:30:49	00:20:52	00:16:33
▲ velvetg Time	00:22:35	00:18:04	00:16:45	00:13:12

Chart 4

For escherichia coli (1): Time in minutes used by velveth, velvetg and the total time of the procedure as a function of the number of thread

Percentage of time gained			
Cores	**velveth**	**velvetg**	**Total Time**
2	11.0%	20.0%	12.4%
4	39.7%	25.8%	29.3%
8	55.0%	41.5%	41.0%

Table 2
For escherichia coli (1): Percentages of gained time for a single to a multithreaded execution.

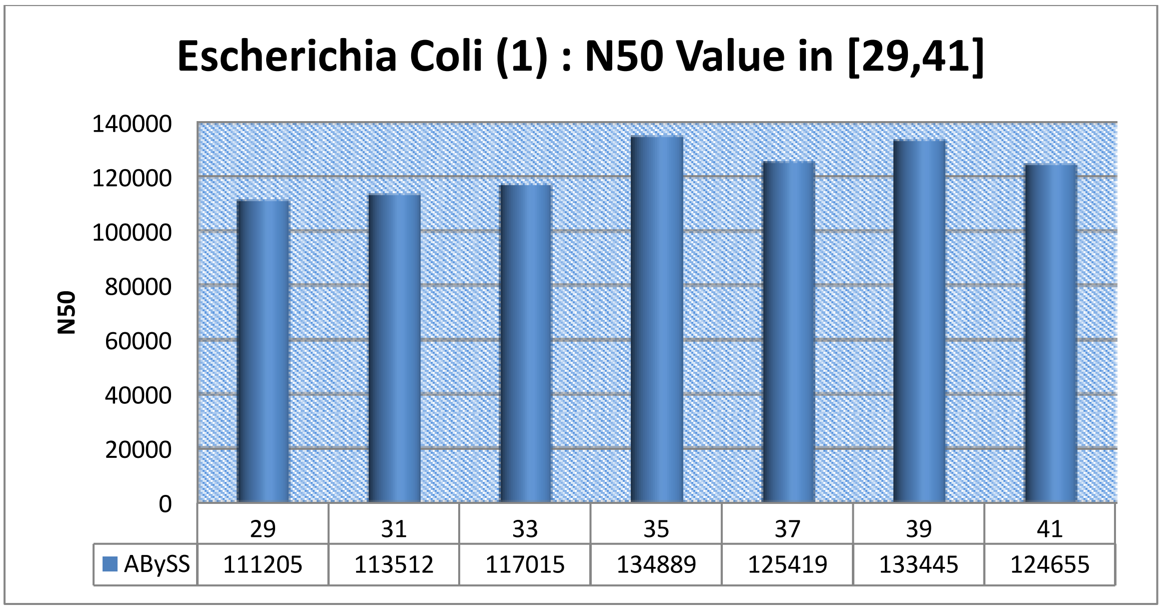
ABySS:

Following the same steps with ABySS in order to define a good k-mer length we had the results shown in Chart 9 and Chart 5.

In first test the best N50 this time was not observed in the same k-mer length with Velvet as in the experiment of staphylococcus aureus. Here the best N50 was achieved with the use of a k-mer length 35. None of the test finished with a missassembly as all the total contigs lengths are smaller than Escherichia coli's genome size. K-mer lengths in 35, 45, 55 have a similar N50 output value. Although in ABySS N50 is not affected by discrepancies, thus we can chose k-mer length 35 based on N50 output as the best assembly for the first test.

In the second test resulting N50 using the odd k-mers lengths around 35 resulted in N50 similar values with again a small peak in 37. These values range from ~100Kbp to ~135Kbp that is a quite small interval. The only think that it can be mentioned here is bigger kmer lengths lead to a fastest procedure. This is because more spurious overlaps between unrelated k-mers are minimized and the error correction step's duration is less.

After the two test made above we proceed to the MPI test. The behavior of time in relation with the used nodes is shown in Chart 8. As is shown again there no a great decrease in time but the results are more visible than with the smaller previously tested genome. In detail time fall gradually from the single node execution till the use of 6 nodes having a decrement in the processing time of about ~33.5% that is similar with the time gained with the use of four nodes in the staphylococcus aureus. Then the gain in time cause by the parallelization slightly disappear while increasing the nodes showing again that for this size of libraries then use of more than 96 CPUs, i.e 6 nodes, is not approapriate in order to achieve better timing.



	29	31	33	35	37	39	41
■ ABySS	111205	113512	117015	134889	125419	133445	124655

Chart 5
Escherichia Coli (1) N50 values in the interval [29,41]

6.5 Escherichia Coli (2)

Velvet:

In this experiment, testing again an Escherichia coli genome were used two pair end fragment libraries and two shortjumping libraries instead of one and one respectively in the first experiment of the same genome. Due to this the preliminary step of shuffling and reverse complementing will add a bigger overhead in the time. The results of this preliminary step are shown below:

Action	Input Size	Time
Shuffle fragment library	2x3.28 GB	0:10:56
Shuffle fragment library	2x3.27 GB	00:11:40
Shuffle short jumping libraries	2x374.53 MB	00:01:17
Shuffle short jumping libraries	2x149.49 MB	00:00:13
Reverse complement shuffled short jumping libraries	749.06 MB	00:01:53
Reverse complement shuffled short jumping libraries	374.53 MB	00:00:21

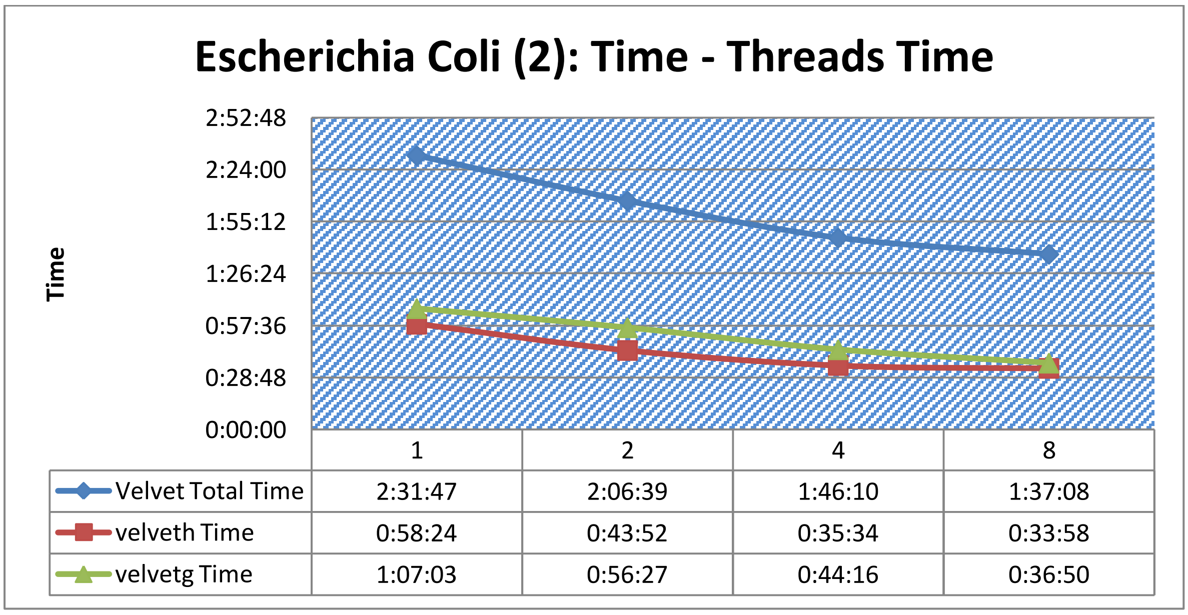
The total time of all the action mentioned above is 26.20 minutes.

The examples made in this test follow the same idea. For k-mer length 25, 35, 45, 55 the results are shown in Chart 10. While for the second test done in this time with a k-mer length of 59 are shown in Chart 7

The results in the first test have the same behavior with the previous experiment of Escherichia coli with smaller values. Again with k-mer lengths 25, 35 the output N50 and the total length of the contigs show that the assembly process failed leading to wrongs results. Then the N50 statistic increase rapidly but with a slight difference of the previous experiment. Previously k-mer lengths 45 and 55 had a similar N50 output but now they differ a lot showing clearly that 55 is an appropriate k-mer length to test this genome.

Having 55 as a reference k-mer length the second test took place. The around odd k-mer lengths produced quite similar results in sense of N50 value with a peak at k-mer length of 59 that is near to the one observed in the previous experiment of Escherichia coli. Even if the N50 does not vary a lot the all the assemblies in the interval [49,53] are missassmblies. The strange in this test is the the highest N50 output is accompanied from the smaller max contig length. This mean that the other decent executions create few big contigs and a big amount of small ones, thing that is verified form the total contigs produced, while the one with the biggest N50 create contigs of nearby lengths, as the N50 value is similar to the max contig length.

The results of the multithreading test done are presented below. As shown the total time that velvet proceeds the data and given the output decreases almost linearly with the use of two and four cores. Then time continues its decrement but this time not more linearly introducing some overhead form copies of memory that is not compensated like before..



	1	2	4	8
Velvet Total Time	2:31:47	2:06:39	1:46:10	1:37:08
velveth Time	0:58:24	0:43:52	0:35:34	0:33:58
velvetg Time	1:07:03	0:56:27	0:44:16	0:36:50

Chart 6

For escherichia coli (2): Time in minutes used by velveth, velvetg and the total time of the procedure as a function of the number of threads

Percentage of time gained			
Cores	**Velveth**	**Velvetg**	**Total Time**
2	13.8%	15.8%	16.6%
4	30.1%	34.0%	30.0%
8	33.3%	45.0%	36.0%

Table 3

For Escherichia coli (2): Percentages of gained time for a single to a multithreaded execution

<u>*ABySS:*</u>

Even the genome is the same tested in the previous experiment the results are different. The addition of more pair end and mate pairs libraries changed the k-mer area that is going to be examined. **Chart 10** and **Chart 7** show the results of the first two tests.

In the first test all k-mers produced reliable outputs as the total contig length is smaller than the examined genome size. Moreover, the N50 values are very similar with a small peak in k-mer length 55 that is much bigger than the one used previously, i.e. 35. This is because the libraries in this example provide more coverage. Higher k-mer means higher specificity in the graph, and this reduces the k-mer coverage that the library provides.

The second test resulted in N50s that differ between them at max 2Kbp. This is a very small range showing that this entire interval is quite reliable with the only exception the k-mer length of 61 that created contigs with total length bigger than 4,639.675bp. Another point to be mention is that the best N50 observed in present three times, one with k-mer length 55, 59 and 61. We discard k-mer length of 61 as produced a missassembly. In order to decide which k-mer length between the remaining two will be chosen we based our choice in the maximum contig length and the chosen one is k-mer with length 59.

Bigger size of libraries for the same genome led to more clear view about the use of MPI. The pass of the procedure from a single node to two has a more obvious slope with a gain in type of about ~31.2%. This percentage was achieved with four and eight nodes respectively in the previous two experiments. Time continues decreasing, but with not big differences, and achieves its lower value with the use of 128 CPUs with a gain in time of more than 50%. In the appendix are shown the run with 144 CPUs and 160 that led to a slower process

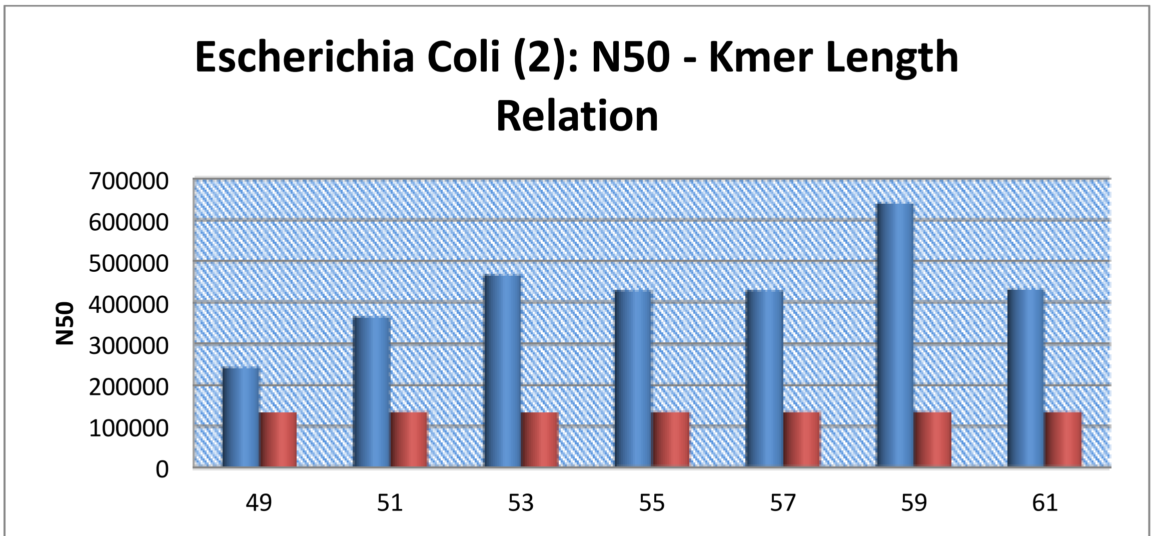
6.6 Rhodobacter sphaeroides

<u>Velvet</u>

Velvet could not run this experiments due lack of RAM.

<u>ABySS</u>

In this last experiment only the first test took place and its results are shown in Chart 8. The thing that compared to the other execution differs a lot in the k-mer that led to the best N50 value. As mentioned, when the given k-mer coverage is big and the genome is small then the best assembly, in sense of N50 output, should be observer with a big k-mer. In this case things are different as there is provided big k-mer coverage related to the sizes of the libraries. The bigger libraries are mate pairs. In these experiments the libraries used for the contiging are small compared to the mate pairs.



	49	51	53	55	57	59	61
🟦 Velvet	243195	364141	466670	428679	428975	640127	430884
🟥 ABySS	133562	134963	133562	135012	134413	135012	135012

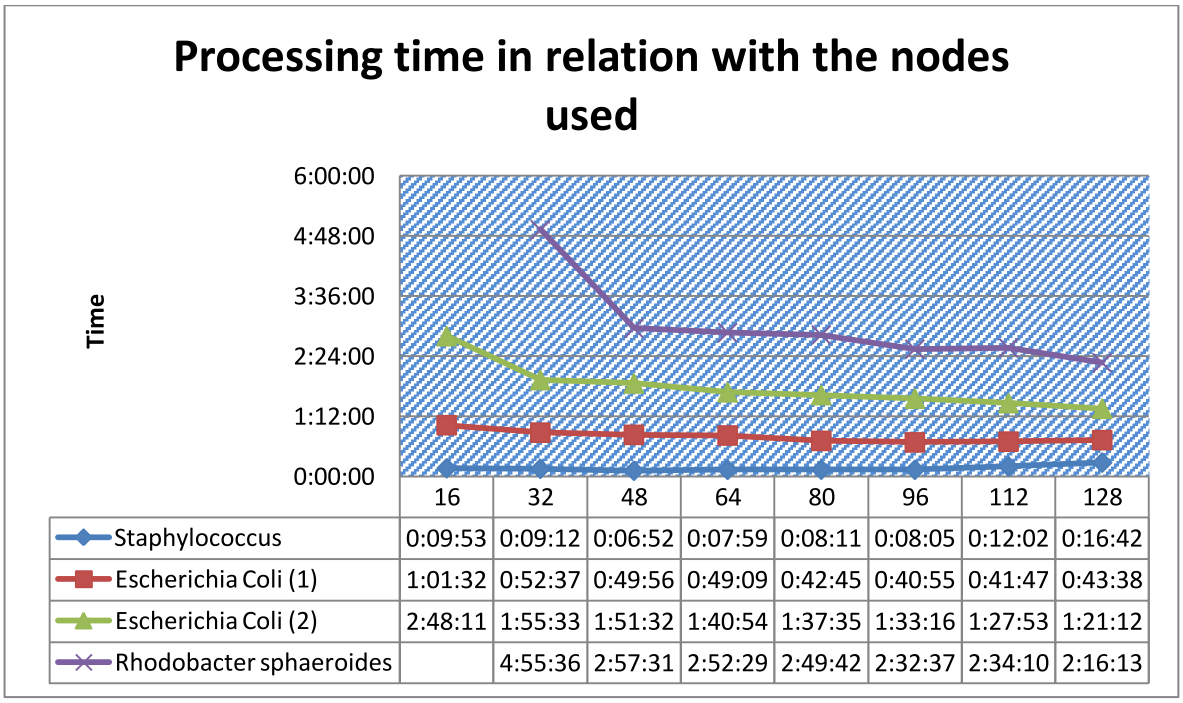
Chart 7

Escherichia Coli (2) N50 values in interval [49,61]

The latter ones take part only in the scaffolding stage trying to merge the previously created contigs. So the k-mer length does not affect significantly the scaffolding stage.

Due to this a much bigger N50 value was produced with a small k-mer length,i.e 25. The other two reliable assemblies with k-mer lengths 25 and 55, as the one with a k-mer value of 45 produced a missassembly, have similar N50 outputs but is limited to the half achieved with a k-mer of length 25.

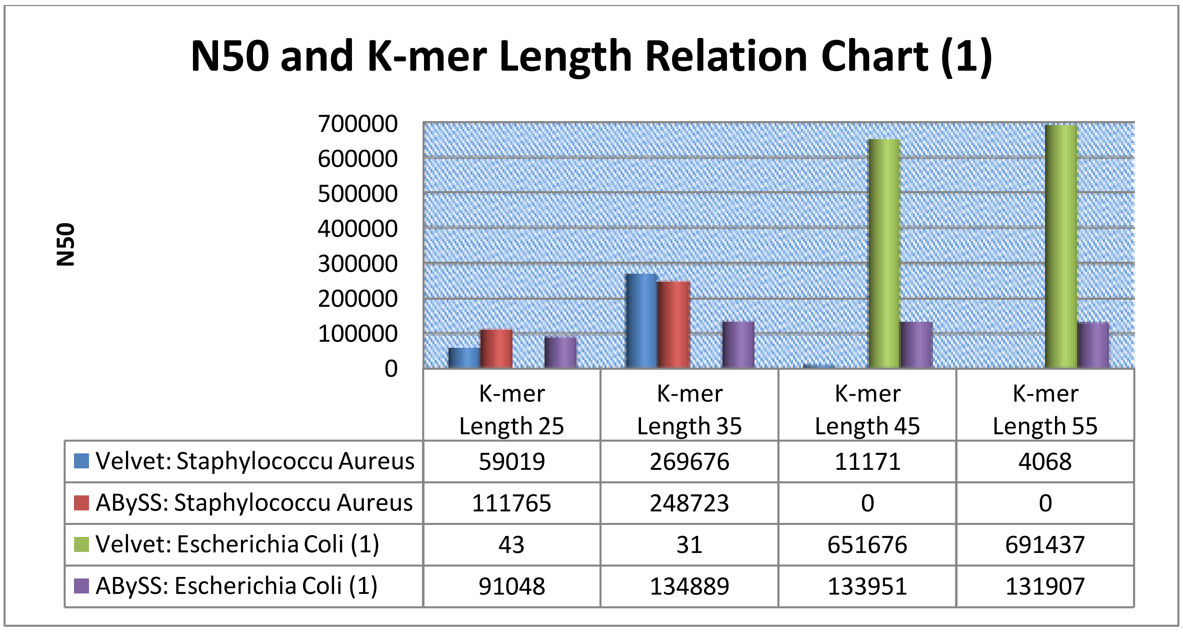
With this k-mer the MPI's results are showin in Chart 8. The thing that must be mentioned here is the memory of one node was not enough to complete the procedure so we cannot have a comparison with a single node execution. Although results here show that the passage from two nodes to three has a biggest decrease in time of about . Then the behavior is similar to the one observer in the other experiments, i.e. not big changes in time. The ideal number of nodes, in sense of faster procedure, became bigger this time arriving at 10. See appendix



	16	32	48	64	80	96	112	128
◆ Staphylococcus	0:09:53	0:09:12	0:06:52	0:07:59	0:08:11	0:08:05	0:12:02	0:16:42
■ Escherichia Coli (1)	1:01:32	0:52:37	0:49:56	0:49:09	0:42:45	0:40:55	0:41:47	0:43:38
▲ Escherichia Coli (2)	2:48:11	1:55:33	1:51:32	1:40:54	1:37:35	1:33:16	1:27:53	1:21:12
✕ Rhodobacter sphaeroides		4:55:36	2:57:31	2:52:29	2:49:42	2:32:37	2:34:10	2:16:13

Chart 8

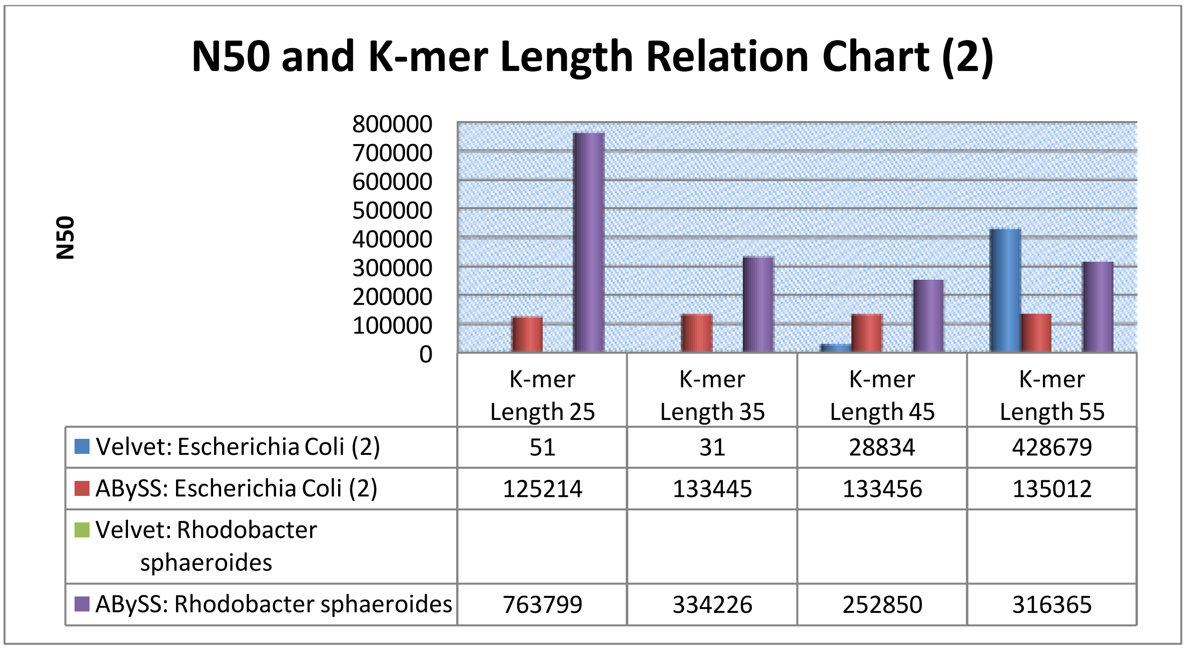
The time benchmarking of the four experiments with the use of a different size of nodes.



	K-mer Length 25	K-mer Length 35	K-mer Length 45	K-mer Length 55
🟦 Velvet: Staphylococcu Aureus	59019	269676	11171	4068
🟥 ABySS: Staphylococcu Aureus	111765	248723	0	0
🟩 Velvet: Escherichia Coli (1)	43	31	651676	691437
🟪 ABySS: Escherichia Coli (1)	91048	134889	133951	131907

Chart 9

N50 in relation with kmer length 25, 35, 45, 55 for Velvet and ABySS in Staphylococus Aureus and Escherichia Coli (1) genomes



	K-mer Length 25	K-mer Length 35	K-mer Length 45	K-mer Length 55
🟦 Velvet: Escherichia Coli (2)	51	31	28834	428679
🟥 ABySS: Escherichia Coli (2)	125214	133445	133456	135012
🟩 Velvet: Rhodobacter sphaeroides				
🟪 ABySS: Rhodobacter sphaeroides	763799	334226	252850	316365

Chart 10

N50 in relation with kmer length 25, 35, 45, 55 for Velvet and ABySS in Escherichia Coli (2) and Rodobacter sphaeroides genomes

6.7 RAM Consumption

<u>Velvet:</u>

The most common way to check Velvet's memory is to use "top" and observe the peak value that the RAM memory will reach. Although this command is disabled in Magerit so we base our memory measurements in a formula given from another benchmark [29]. This formula gives an estimation of the peak RAM consumption in velveth and velvetg based on the millions of sequences being assembled. In detail,

for velveth

$mem = 1.7N + 2.7$

while for velvetg

$mem = 2.2N - 4.0$

where mem is in GB and N is the number of millions of sequences

Applying this formulas to our libraries we the following summarized results:

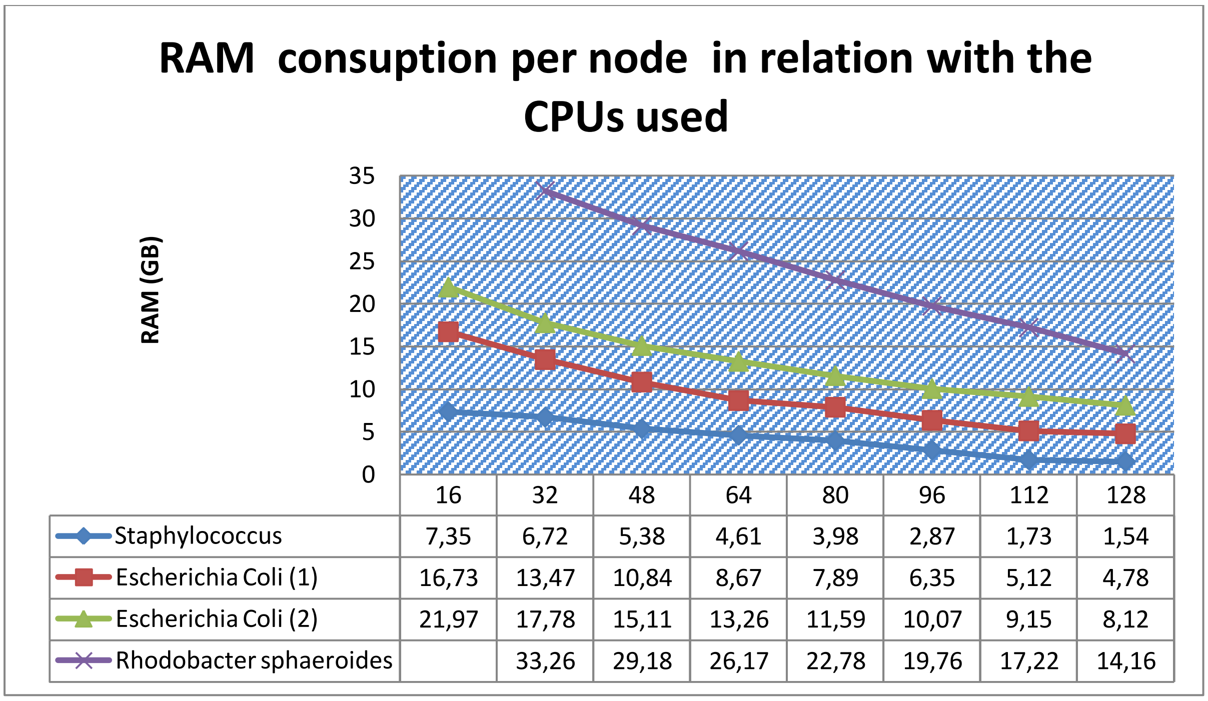
	Staphylococcus Aureus	**Escherichia Coli (1)**	**Escherichia Coli (2)**	**Rhodobacter Sphaeroides**
velveth RAM	10.84 GB	23.04 GB	51.847 GB	86.47 GB
velvetg RAM	6.53 GB	22.33 GB	59.58 GB	104.35 GB

Here it is clear that the Rhodobacter Sphaeroides cannot be assembled in a node of 96 GB of RAM.

A thing to notice is for a small amount of assembled sequences velveth needs more memory while things change as velvetg consumes more memory.

<u>ABySS:</u>

In ABySS the memory is reported in the output log file if in the execution line is included the verbose output parameter, i.e. *v=-v*. For every assembled MPI test made in the 3 genomes the values of the memory consumed per node are summarized in the following chart.

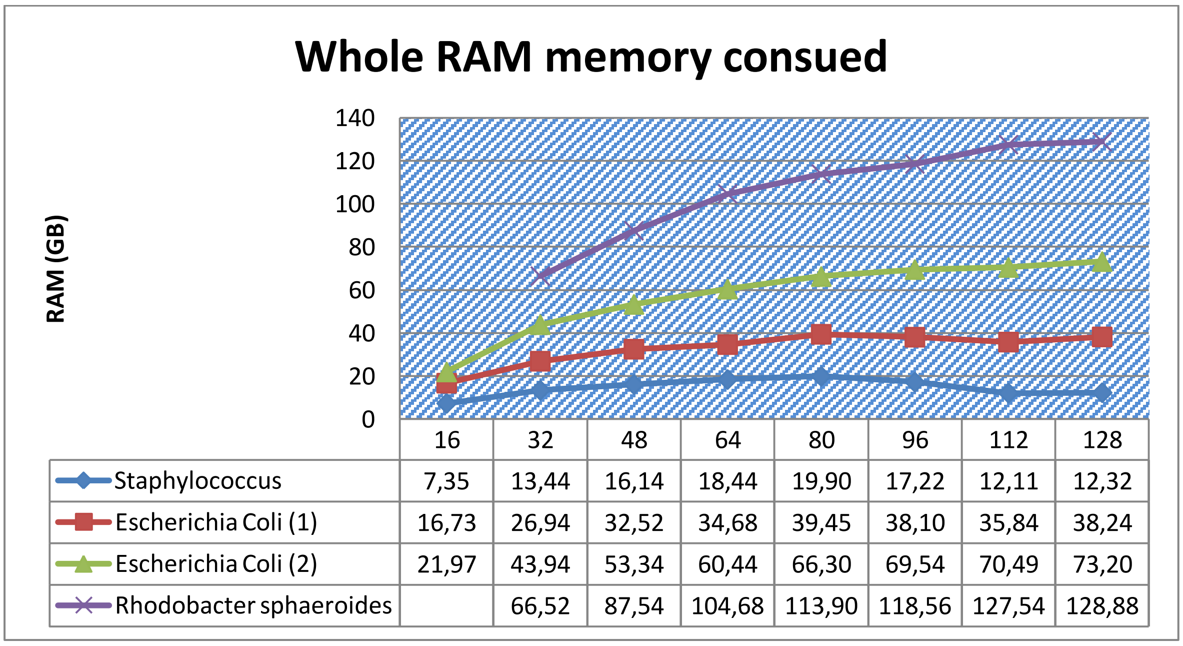


	16	32	48	64	80	96	112	128
Staphylococcus	7,35	6,72	5,38	4,61	3,98	2,87	1,73	1,54
Escherichia Coli (1)	16,73	13,47	10,84	8,67	7,89	6,35	5,12	4,78
Escherichia Coli (2)	21,97	17,78	15,11	13,26	11,59	10,07	9,15	8,12
Rhodobacter sphaeroides		33,26	29,18	26,17	22,78	19,76	17,22	14,16

Chart 11

Peak memory used in every nodes one different MPI execution for the 4 experiments

In this **Chart 11** and **Chart 12** we can see that increasing the nodes memory is shared between them although the total memory consumption it increases. Obviously for big genomes the aggregation of memory that ABySS provides make it suitable for the assembly of large genomes.



	16	32	48	64	80	96	112	128
Staphylococcus	7,35	13,44	16,14	18,44	19,90	17,22	12,11	12,32
Escherichia Coli (1)	16,73	26,94	32,52	34,68	39,45	38,10	35,84	38,24
Escherichia Coli (2)	21,97	43,94	53,34	60,44	66,30	69,54	70,49	73,20
Rhodobacter sphaeroides		66,52	87,54	104,68	113,90	118,56	127,54	128,88

Chart 12

Peak total RAM memory consumed in the MPI tests for all the experiments.

6.8 Comparison

The good point that the two assembler have is that they have been paralized and the can share the procedure reducing the computational time. Velvet is parallelized with OpenMP that limit him to one node while AbySS is parallelized with MPI so lots of nodes can be used.

From the results achieved above we can see that for small genomes Velvet is faster than AbySS but while the genome gets bigger AbySS can achieve better timing. A disadvantage of Velvet is the huge amount of RAM that needs for an assembly procedure. The ability of AbySS to share the memory makes it suitable for big genomes while for Velvet this is not valid. Moreover, multithreading is not deterministic - threads will get different data at different times depending on the scheduler and load of your machine etc. The scaffolding step in particular will give different answers each time someone runs it. This can cause problem that can be only treated with executing the assembly with one thread. Although this solution for big genomes can be very time expensive, more than running two times the same assembly and check the differences in the N50 value.

Most of the assemblies done in this gave better N50 values with the use of Velvet. Althought ABySS is reported that is suitable for bigger k-mer lengths than the one used.

Another point that the two assembler have in common is that with bigger k-mer lengths the assembly procedure was sorter. This is cause by the k-mer coverage. Bigger length of a k-mer mean higher specifity in the graph that limits the coverage provided leading to shorter assemblies.

Appendix:

VELVET TESTS

Staphylococcus

K-mer Length 25, 35, 45, 55

N. of threads	Velveth Time	Velvetg Time	Shuffling Time	Total Time	N50	Max	Total
8	00:01:50	00:03:18	0:01:08	00:06:16	59019	273046	2932726
8	00:01:44	00:02:22	0:01:08	00:05:14	269676	778608	2831215
8	00:01:37	00:01:55	0:01:08	00:04:40	11171	54726	2832115
8	00:01:33	00:01:41	0:01:08	00:04:22	4068	20606	2772458

K-mer Length 29, 31, 33, 35, 37, 39, 41

N. of threads	Velveth Time	Velvetg Time	Shuffling Time	Total Time	N50	Max	Total
8	00:01:51	00:02:34	0:01:08	00:05:33	213338	342467	2915661
8	00:01:43	00:02:46	0:01:08	00:05:37	269602	596270	2908076
8	00:01:46	00:02:26	0:01:08	00:05:20	298823	797290	2908045
8	00:01:45	00:02:27	0:01:08	00:05:20	327929	823771	2832215
8	00:01:47	00:02:20	0:01:08	00:05:15	270361	800965	2893877
8	00:01:42	00:01:53	0:01:08	00:04:43	20655	73780	2839881
8	00:01:38	00:02:04	0:01:08	00:04:50	18359	58983	2834780

Multithreading K-mer Length 35

N. of threads	Velveth Time	Velvetg Time	Shuffling Time	Total Time	N50	Max	Total
1	0:01:51	0:02:33	0:01:08	0:05:32	322690	823974	2834459
2	0:01:40	0:02:25	0:01:08	0:05:13	200366	825843	2833424
4	0:01:44	0:02:16	0:01:08	0:05:08	373070	826124	2837046
8	0:01:45	0:02:26	0:01:08	0:05:19	278486	825878	2835915

Preparation

Action	Input Size	Time
Shuffle fragment libraries	2x140 MB	0:00:26
Shuffle short jumping libraries	2x 161.41 MB	0:00:38
Reverse complement shuffled short jumping libraries	322.82 MB	0:00:04

Escherichia Coli (1)

K-mer Length 25, 35, 45

N. of threads	Velveth Time	Velvetg Time	Shuffling Time	Total Time	N50	Max	Total
8	00:18:41	11:25:37	0:09:42	11:54:00	43	7406	85717329
8	00:18:17	00:32:31	0:09:42	1:00:30	31	259	18813171
8	00:18:20	00:17:48	0:09:42	0:45:50	651676	1299247	4625290
8	00:17:26	00:13:17	0:09:42	0:40:25	691437	2183079	4608739

K-mer Length 49, 51, 53, 55, 57, 59, 61

N. of threads	Velveth Time	Velvetg Time	Shuffling Time	Total Time	N50	Max	Total
8	00:23:00	00:16:50	0:09:42	00:49:32	694097	1201535	4614655
8	00:19:09	00:14:30	0:09:42	00:43:21	651262	1526313	4612360
8	00:17:01	00:14:00	0:09:42	00:40:43	676458	788138	4609069
8	00:16:40	00:13:44	0:09:42	00:40:06	1131731	1249922	4606350
8	00:16:47	00:13:30	0:09:42	00:39:59	1193565	1282816	4605441
8	00:17:31	00:13:35	0:09:42	00:40:48	691638	1194669	4609664
8	00:16:22	00:13:29	0:09:42	00:39:33	691763	1193905	4608824

Multithreading K-mer Length 57

N. of threads	Velveth Time	Velvetg Time	Shuffling Time	Total Time	N50	Max	Total
1	00:34:37	00:22:35	00:09:42	01:06:54	770481	1282963	4605230
2	00:30:49	00:18:04	00:09:42	00:58:35	692118	1198957	4605943
4	00:20:52	00:16:45	00:09:42	00:47:19	1193977	1281907	4604097
8	00:16:33	00:13:12	00:09:42	00:39:27	1193290	1283203	4607350

Preparation

Action	Input Size	Time
Shuffle fragment libraries	2x2.50 GB	00:07:35
Shuffle short jumping libraries	2x374.53 MB	00:01:02
Reverse coplement shuffled short jumping libraries	749.06 MB	00:01:05

Escherichia Coli (2)

K-mer Length 25, 35, 45

N. of threads	Velveth Time	Velvetg Time	Shuffling Time	Total Time	N50	Max	Total
8	01:07:48	04:29:42	0:26:20	06:03:50	51	9791	91517267
8	01:06:33	02:37:47	0:26:20	04:10:40	31	1002	20265796
8	01:01:06	01:47:57	0:26:20	03:15:23	28834	114108	4555928
8	00:57:21	01:23:26	0:26:20	02:47:07	428679	831525	4637928

K-mer Length 49, 51, 53, 55, 57, 59, 61

N. of threads	Velveth Time	Velvetg Time	Shuffling Time	Total Time	N50	Max	Total
8	00:58:32	01:47:14	0:26:20	03:12:06	243195	934904	4682775
8	01:02:20	01:29:18	0:26:20	02:57:58	364141	1126687	4666132
8	01:02:43	01:34:35	0:26:20	03:03:38	466670	767760	4649613
8	00:58:40	01:14:46	0:26:20	02:39:46	428679	831525	4637928
8	01:00:26	01:10:59	0:26:20	02:37:45	428975	833457	4637823
8	00:59:57	01:09:35	0:26:20	02:35:52	640127	695659	4634775
8	00:54:54	01:03:56	0:26:20	02:25:10	430884	1015295	4635992

Multithreading K-mer Length 59

N. of threads	Velveth Time	Velvetg Time	Shuffling Time	Total Time	N50	Max	Total
1	0:58:24	1:07:03	0:26:20	2:31:47	640127	695659	4634614
2	0:43:52	0:56:27	0:26:20	2:06:39	640141	696019	4635748
4	0:35:34	0:44:16	0:26:20	1:46:10	573051	658856	4634199
8	0:33:58	0:36:50	0:26:20	1:37:08	504975	658698	4634456

Preparation

Action	Input Size	Time
Shuffle fragment library	2x3.28 GB	0:10:56
Shuffle fragment library	2x3.27 GB	00:11:40
Shuffle short jumping libraries	2x374.53 MB	00:01:17
Shuffle short jumping libraries	2x149.49 MB	00:00:13
Reverse complement shuffled short jumping libraries	749.06 MB	00:01:53
Reverse complement shuffled short jumping libraries	374.53 MB	00:00:21

ABySS TEST

Staphylococcus

K-mer Length 25, 35, 45, 55

Time	N50	Max	Total
20:38:00	111765	364009	3135516
0:09:03	248723	459937	2838175
	0		
	0		

K-mer Length 29, 31, 33, 35, 37, 39, 41

Time	N50	Max	Total
0:18:36	160477	342330	3045194
0:11:04	175086	341343	3082876
0:16:33	322274	947958	2855212
0:09:36	248723	459937	2837746
0:08:02	566235	1300164	2836923
	0	0	0
	0	0	0

MPI testing K-mer Length 37

Cpus	Nodes	Time	N50	Max	Total
16	1	0:09:55	566235	1300155	2835797
32	2	0:08:36	566235	1300152	2835794
48	3	0:06:52	566235	1300148	2835790
64	4	0:07:59	566235	970692	2836923
80	5	0:08:11	566235	1300164	2836923
96	6	0:08:05	566235	970711	2822323
112	7	0:12:02	566235	970708	2822320
128	8	0:16:42	566235	970708	2822323

Escherichia Coli (1)

K-mer Length 25, 35, 45

Time	N50	Max	Total
1:20:12	91048	246339	4563829
1:03:16	134889	413898	4585389
0:59:48	133951	414696	4606922
0:49:37	131907	356831	4615810

K-mer Length 29, 31, 33, 35, 37, 39, 41

Time	N50	Max	Total
1:07:42	111205	413900	4574950
1:02:18	113512	413900	4571499
1:06:49	117015	413909	4571351
1:03:46	134889	413898	4585389
0:58:21	125419	413906	4583613
0:57:22	133445	414029	4592375
0:53:12	124655	414156	4588461

MPI testing K-mer Length 35

CPUs	Nodes	Time	N50	Max	Total
16	1	1:01:32	134889	413898	4585386
32	2	0:52:37	134889	413898	4585386
48	3	0:49:56	124635	413898	4585714
64	4	0:49:09	134889	413898	4585389
80	5	0:42:45	134889	413898	4585389
96	6	0:40:55	134889	413898	4585389
112	7	0:41:47	134889	413898	4585389
128	8	0:43:38	134889	413898	4585389
144	9	0:45:33	134889	413898	4585389
160	10	0:54:19	134889	413898	4585389

Escherichia Coli (2)

K-mer Length 25, 35, 45

Time	N50	Max	Total	K-mer
1:12:05	125214	357124	4595528	25
1:09:36	133445	415288	4619925	35
1:07:46	133456	357669	4619536	45
1:02:03	135012	414992	4630089	55

K-mer Length 49, 51, 53, 55, 57, 59, 61

Time	N50	Max	Total	K-mer
1:02:55	133562	415639	4623136	49
1:02:27	134963	414899	4622255	51
1:02:34	133562	414989	4625701	53
1:02:36	135012	414992	4630172	55
1:07:51	134413	414992	4625833	57
1:01:01	135012	415091	4631019	59
0:58:54	135012	415091	4734791	61

MPI testing

CPUs	Nodes	Time	N50	Max	Total
16	1	2:48:11	135012	415091	4734791
32	2	1:55:33	135012	415091	4734791
48	3	1:51:32	135012	415091	4734791
64	4	1:40:54	135012	415091	4734791
80	5	1:37:35	135012	415091	4734791
96	6	1:33:16	135012	415091	4734791
112	7	1:27:53	135012	415091	4734791
128	8	1:21:12	135012	415091	4734791
144	9	01:21:32	135012	415091	4734791
160	10	01:23:11	135012	415091	4734791

Rhodobacter sphaeroides

K-mer Length 25, 35, 45

Time	N50	Max	Total	K-mer	
02:48:27	763799	1749633	4474272	25	417060

2:04:43	334226	1750657	4554389	35	417042
1:53:00	252850	588316	4708867	45	417031
1:45:50	316365	770679	4594987	55	417072

MPI testing

CPUs	Nodes	**Time**	**N50**	**Max**	**Total**	**K-mer**
8	1	12:01:22	763799	1749633	4473923	25
16	1	11:45:38	763799	1749633	4473923	25
32	2	8:55:36	763799	1749633	4473923	25
48	3	2:57:31	763799	1749633	4473923	25
64	4	2:52:29	763799	1749633	4473923	25
80	5	2:49:42	763799	1749633	4473923	25
96	6	2:32:37	763799	1749633	4473951	25
112	7	2:34:10	763799	1749633	4473951	25
128	8	2:16:13	763799	1749633	4473951	25
144	9	2:15:48	763799	1749633	4473951	25
160	10	2:10:42	763799	1749633	4473951	25
176	11	2:11:15	763799	1749633	4473951	25
192	12	2:27:50	763799	1749633	4473951	25

Bibliography

[1] Genomes. DeWeerdt, S. E. (2003, 1 15). From Genome News Network: http://www.genomenewsnetwork.org/resources/whats_a_genome/Chp1_1_1.shtml#genome1

[2] Genes. (n.d.). From News Medical: http://www.news-medical.net/health/Genes-What-are-Genes.aspx

[3] Genetics Home Reference. (n.d.). From Genetics Home Reference: http://ghr.nlm.nih.gov/

[4] JGI Genome Portal. (n.d.). From JGI Genome Portal: http://genome.jgi-psf.org/help/index.html

[5] Guía del Usuario de Magerit-Ejecución de trabajos. n.d. http://docs.cesvima.upm.es/magerit- user-guide/es/magerit/scheduler.html#magerit-scheduler (accessed 2012)

[6]]Whole genome sequencing. n.d. http://en.wikipedia.org/wiki/Full_genome_sequencing (accessed 2012).

[7] Sovic, I. (s.f.). Approaches to DNA de novo assembly. Zagreb: Ruder Boškovic Institute

[8]Shotgun sequencing. n.d. http://en.wikipedia.org/wiki/Shotgun_sequencing

[9] Compeau, P., Pevzner, P., & Tesler, G. (2011). How to apply de Bruijn graphs to genome assembly. Nature Biotechnology 29, 987–991 (2011) doi:10.1038/nbt.2023

[10] Research. n.d. http://www.phrap.org/index.html (accessed 2012).

[11] De Bruijn Graphs. n.d. http://www.homolog.us/blogs/2011/07/29/de-bruijn-graphs-ii/ (accessed 2012).

[12] Human Genome Project Information. (2009, 10 9). From Human Genome Project Information: http://www.ornl.gov/sci/techresources/Human_Genome/project/benefits.shtml

[13] Next Generation Sequencing. (n.d.). From Eurofins mwg Operon: http://www.eurofinsdna.com/service-corner/faqs-products-services/faqs-genome-sequencing/questions-on-genome-sequencing-services/what-is-a-mate-pair-library.html

[14] Zerbino D., Birney E. "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs." (Genome Research) 2008. 18: 821-829. doi: 10.1101/gr.074492.107

[15] Jared T. Simpson, Kim Wong, Shaun D. Jackman, Jacqueline E. Schein, Steven J.M. Jones, İnanç Birol. "ABySS: A parallel assembler for short read sequence data" (Genome Research) 2009. 19: 1117-1123. doi: 10.1101/gr.089532.108

[16] Zerbino DR, McEwen GK, Margulies EH, Birney E (2009) "Pebble and Rock Band: Heuristic Resolution of Repeats and Scaffolding in the Velvet Short-Read de Novo Assembler". PLoS ONE 4(12): e840. doi:10.1371/journal.pone.0008407

[17] Queueing System . (2012, 6 2). From Freie Universitat Berlin: http://www.zedat.fu-berlin.de/Compute/EN/SorobanQueueingSystem

[18] Jones, M. T. (2012, 5 22). Optimizing resource management in supercomputers with SLURM. From developersWork: http://www.ibm.com/developerworks/library/l-slurm-utility/

[19] Pop., M., Salzberg, S. L., & Shumway, M. (n.d.). Genome Sequence Assembly:Algorithms and Issues. Yhe Insitute of Genomic Research. July 2002 (vol. 35 no. 7)

[20] Ahmed, M., Ahmad, I., & Khan, S. U. (2010). A comparative analysis of parallel computing approaches for genome assembly. 2011 Mar;3(1):57-63. Epub 2011 Mar 3.

[21] Can Alkan, Saba Sajjadian & Evan E Eichler. Limitations of next-generation genome sequence assembly. Nature Methods 8, 61–65 (2011) doi:10.1038/nmeth.1527

[22] Dent A. Earl, Keith Bradnam, John St. John, Aaron Darling, Dawei Lin, Joseph Faas, Hung On Ken Yu, Buffalo Vince, Daniel R. Zerbino, Mark Diekhans, Ngan Nguyen, Pramila Nuwantha, Ariyaratne Wing-Kin Sung, Zemin Ning, Matthias Haime, Jared T. Simpson, Nuno. "Assemblathon 1: A competitive assessment of de novo short read assembly methods." (Genome Research) 2011. doi: 10.1101/gr.126599.111

[23] Michael C. Schatz, Arthur L. Delcher, Steven L. Salzberg. "Assembly of large genomes using second-generation sequencing." (Genome Research) 2010 Sep;20(9):1165-73. Epub 2010 May 27. doi: 10.1101/gr.101360.109

[24] Jason R. Miller, Sergey Koren, Granger Sutton. Assembly algorithms for next-generation sequencing data. 2010 Jun;95(6):315-27. Epub 2010 Mar 6..

[25] Zhang W, Chen J, Yang Y, Tang Y, Shang J. A Practical Comparison of De Novo Genome Assembly Software Tools for Next-Generation Sequencing Technologies. PLoS ONE 6(3): e17915. doi:10.1371/journal.pone.0017915

[26] Noonan, Jim. "Sequence Assembly and Alignment." n.d. http://www.gersteinlab.org/courses/452/09-spring/pdf/SeqAssembly.pdf (accessed 2012).

[27] abyss-pe. n.d. http://manpages.ubuntu.com/manpages/precise/en/man1/abyss-pe.1.html (accessed 2012).

[28] Genome. n.d. http://www.broadinstitute.org/education/glossary/genome (accessed 2012).

[29] Jennifer Commins, Christina Toft and Mario A. Fares. “Computational Biology Methods and Their Application to the Comparative Genomics of Endocellular Symbiotic Bacteria of Insects”. Biological Procedures Online, 2009, Volume 11, Number 1, Pages 52-78

[30] Velvet performance in the computing service of the UPV/EHU. June 10, 2012. General Service of Informatics Applied to the Research

[31] Daniel Zerbino. Velvet Manual - version 1.1. August 29, 2008

[32] Steven L. Salzberg and James A. Yorke. Beware of mis-assembled genomes.Bioinformatics (2005) 21 (24): 4320-4321