Studies of Ab Initio Genome Assembly Algorithms In HPC Platforms

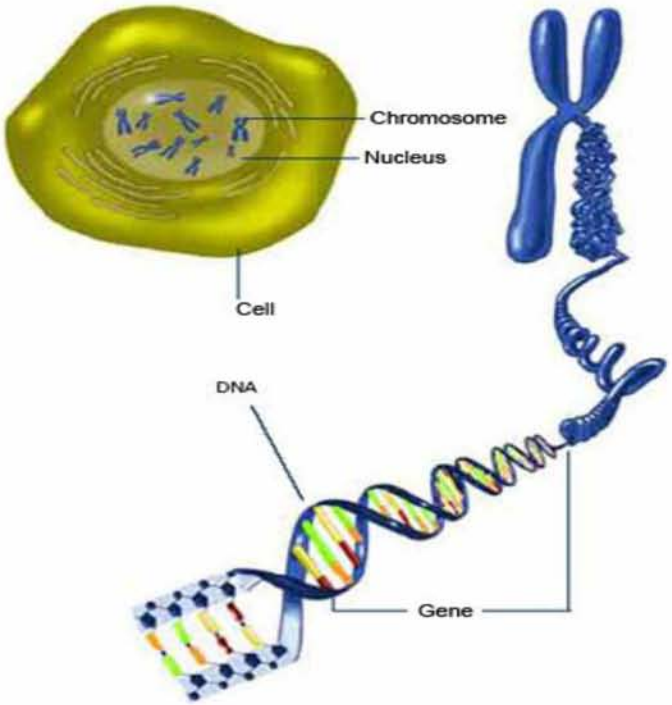
Nikolaos Ioannidis
niioanni@gmail.com

Madrid
7/13/2012

Contents

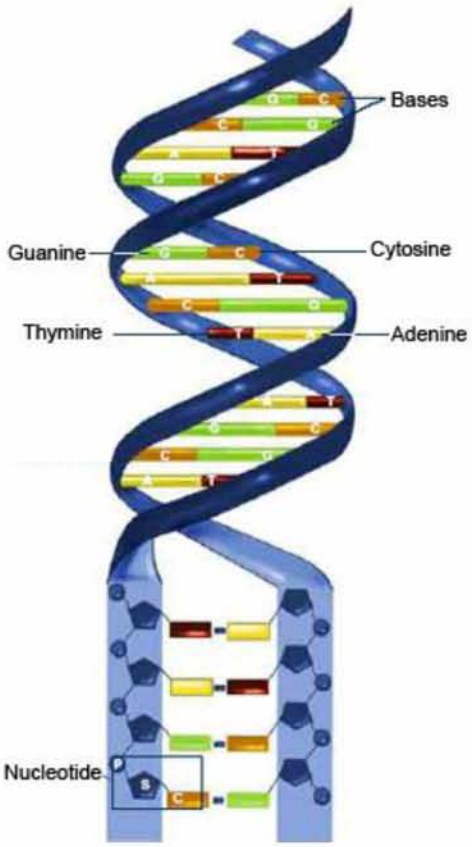
1. Genome Description and Characteristics

All of the biological information of every organism, including humans, is included in the genome. This biological information that is contained in the genome is encoded in DNA or, for many types of virus, in RNA. The genome is divided into discrete units called genes. Except from the genes includes the non-coding sequences of the DNA/RNA. With the non-coding DNA are described the components of an organism's DNA sequences that do not encode for protein sequences and because much of this DNA does not have important information and no known biological functions referred as "junk DNA". Non-coding RNAs are functional RNA molecules that are not translated into protein.



At the picture of the left is seen the different parts of the cell, where the chromosomes are located and the part that is considered as genome.

A chromosome is a single piece of coiled DNA that contains many genes and other nucleotide sequences. The chromosomes contain the associated proteins, which serve to pack the DNA and control its functions. A chromosome is a structure of DNA which is organized for the purpose of the above tasks.

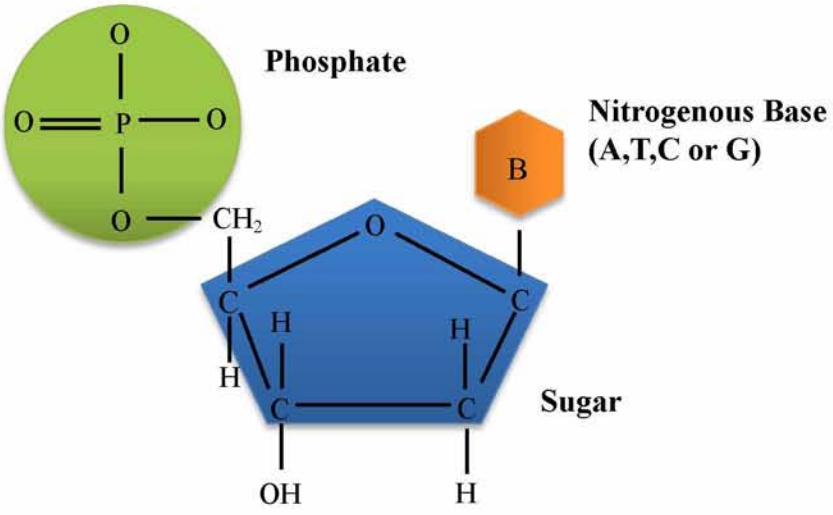


The double stranded nature of the DNA.

The DNA strands consisted of the four bases Adenine, Guanine Thymine Cytosine.

Adenine is creating bonds and connected only with Thymine and Guanine only with Cytosine.

Nucleotides are the basic building blocks of DNA, consisting of a sugar (deoxyribose) with a phosphate group attached to it, and a base. The base of a nucleotide can be one of the following: adenine, cytosine, guanine and thymine. In graphical or textual representations of DNA molecules, nucleotides are commonly denoted through the abbreviations of their base names A, T, C and G. In this case, the DNA can be written as the sequence of these four letters. Nucleotide A from one strand of the DNA always bonds to a nucleotide of the other strand of the DNA, and vice versa. For this reason A and T nucleotides are called complements. The same applies for the C and the G nucleotides.



2. Description of Genome Assembly

Genome assembly refers to the process of taking a large number of short DNA sequences and putting them back together to create a representation of the original chromosomes from which the DNA originated.

2.1 The Procedure

The goal of Genome assembly is to create a representation of the original chromosomes of the living organism from which the DNA originated. This living organism could be an animal, a plant, a bacterium, a virus, a fungus, an archaean or a protist. It is the process of taking a large number of short DNA sequences and putting them together in order to create the above representation. The point of this process is the determination of the complete genome sequence and important genome encoded features.

In order to see how the overall DNA sequence is produced, the first step is to determine the method that the DNA sequences are extracted and given to genome assemblers as inputs. So, let's take a look at the basic and primal method and the parts of it, so that a general idea will be produced before the later analysis. This method that is utilized for DNA sequencing is known as the Sanger method or chain termination. This method can only be used for shorts strands that their size is 100 up to 1000 basepairs. This is an extremely important limitation, since the DNA can be up to a few billion nucleotides long.Therefore, the longer sequences must be subdivided so that smaller fragments are going to be produced, which can be subsequently reassembled and give the overall sequence. The principal method which is used for sequencing the entire genome by dividing it in smaller fragments is called whole genome shotgun (WGS) sequencing.

The shotgun sequencing, or shotgun cloning, is named like this because of the random firing pattern of a shotgun. With this method the DNA is randomly fragmented into numerous small segments, which after are sequenced by the sequencing machines in order to produce the final reads that are going be used as inputs by the genome assemblers. The shotgun method a preliminary step at the genome sequencing, is applied for several rounds in multiple copies of the genome which are extracted from a single one template. Each copy is sheared into random fragments. Next from these random fragments are selected the ones that are size appropriate. Continuing from the size fractioned fragments the sequencing machine is producing the reads. There are several types of sequencing machines using different methods, but in this case the Sanger machines use chain termination to produce the reads. So, in this way, multiple reads, that are overlapping, are obtained in order when combined to give the original full genomic sequence.

Multiple copies of the genome

Sheared random fragments

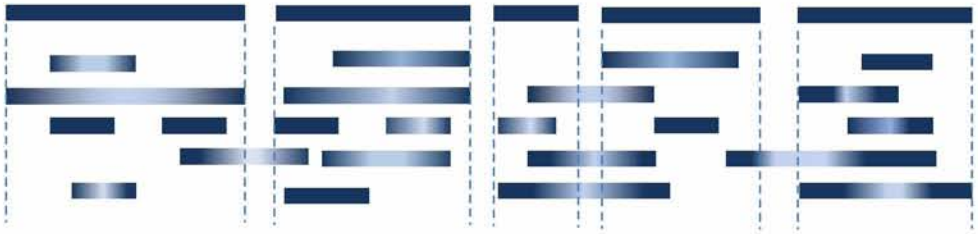
Size fractioned fragments

Reads

These reads are then given as input to automated sequencing machines, the genome assemblers. The reads in fact are sequences of a certain number of nucleotides or bases. With bases, as explained before, is meant adenine(A), cytosine(C), guanine(G)

and thymine(T). Since this information is known, by the meaning that the reads have been given as an input, the algorithm works on this information and attempts to merge them. The reads that overlap can be merged together and form the contigs. The contigs are sets of overlapping reads, which when are align together are forming a consensus region of DNA. This process continues and combining the information of the sequenced contigs are formed the scaffolds. The scaffolds, or super contigs, are a portion of genome sequence, composed of contigs and gaps. The gaps occur from errors at the assembly procedure.

Contigs



Scaffolds

The genome assembly algorithms work in simple steps but really difficult ones when it comes to actual implementation. The genome assembly is a problem with great difficulty and it becomes more difficult because of the abundance in many genomes of sequences that are identical. These regions can occur in thousands different locations and from completely different parts.

Before continuing with an example to make clearer the problem, let's see the distinction between the genomes, according to the size of them

- Small, size is about a few Megabases,
  e.g. bacterial genomes
- Medium, size varies among several hundred Megabases
  e.g. lower plant genomes
- Large, size ranges in Gigabases
  e.g. mammalian and plant genomes

2.2 Examples

The problem of sequence assembly can be compared to taking many copies of a book, passing them all through a shredder, and piecing the text of the book back together just by looking at the shredded pieces. Besides the obvious difficulty of this task, there are some extra practical issues: the original may have many repeated paragraphs, and some shreds may be modified during shredding to have typos. Excerpts from another book may also be added in, and some shreds may be completely unrecognizable.

a) A set of DNA fragments that are cut from multiple copies of the genome sequence

```
ATGA        ATG         ATGATC
TCGA        ATCG        GACAGTA
CAGTA       ACAGTA
```

b) Reads, randomly selected DNA fragments, subject to noise

```
R1: ATGA        R3: ATCG        R5:ATGATC
R2: TCGA        R4: ACAGTA      R6:GACAGTA
```

c) Construct the sequence by identifying overlaps between reads

```
R1: A T G A
R5: A T G A T C
R3:       A T C G
R2:         T C G A
R6:               G A C A G T A
R4:                 A C A G T A
                  ------------>
S : A T G A T C G A C A G T A
    ----------------------------
```

the sequence deduced from the overlaps

In a general way, the genome assembler's job is to put the reads in the proper order by comparing each read to every other with the criteria of how the different reads are overlapping. It would be really easy if the reads that overlap belong next to each other in the final sequence, but over 30% of sequence that the genome contains is repeated numerous times. By this, a repeat of reads could exist between fragments that are millions base pairs apart in the genome. The given output of an assembler is a collection of large stretches of the genome that are put correctly together.

2.3 Aims of Genome Assembly

The aim of genome sequencing is to extract and gain information about the complete set of genes in the examined genome sequence. Also the representation of each genomic sequence in a single scaffold, however this is not always possible. Moreover, by the time the scientist have a clearer image about the role of the non-coding DNA, or "junk DNA", and it is really important to have a good background, by the meaning of a complete genome sequence, that will work as a base reference for the understanding of the genetics and the biology of any given organism. In addition, projects of genome assembly don't aim only in the determination of the DNA sequence but also in gene prediction. This part refers to set the bases in order to find where the genes are in a genome and what those genes do. Also, the produced sequences that are outcome of the genome assembler's are used in genome browsers.

The genome browsers web are interfaces connected to databases that contain sequences produced by the various genome sequencing projects and the related notes. Through them is possible to study the anatomy of the genomes at various degrees of detail, until the final sequence, displaying at the same time all the structural and functional characteristics available for that section of the genome. In addition, for each stretch of DNA is possible to display mapping data, where available.

3. Categorization

The genome assemblers divide in several categories, depending on nthe input data, the existence of a genome reference, how they handle and build the graph of the assembly.

3.1 Categorization by Existence of Reference

Two main categories can be distinguished in the sequence assembly:

1. De-novo: short reads are assembled to create full-length contiguous sequences called contigs, followed by the process of correctly ordering contigs into scaffolds, without having a reference genome

2. Mapping: reads are assembled with reference at an existing backbone sequence, the sequence that is builded at the end it is not necessary to be identical to reference sequence. The pre-existing reference genome sequence can be used to align the reads.

The de novo assemblies are slower and need more memory than the mapping assemblies, and the prior reason for this is that in de novo every read need to compare with every other read, but the importance of de novo assembly is the creation and production of new sequence assemblies for genomes that previously were not assembled or characterized. This kind of assemblers is going to be evaluated and concern us.

3.2 Categorization by Sequencing Technology

3.2.1 Reads Analysis

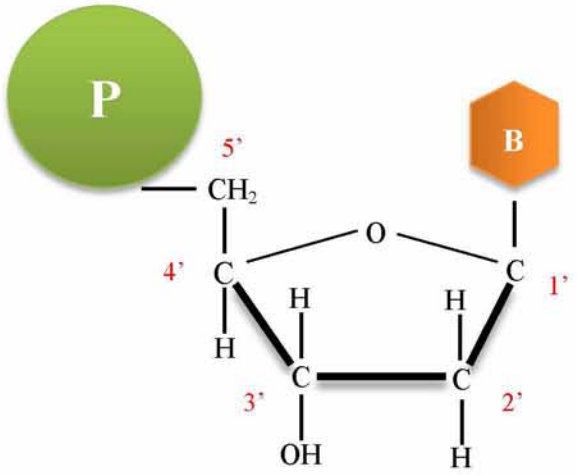
So far we made clear about sequence assembly that is a problem that refers to the alignment and merge of DNA fragments, or reads, of a DNA sequence that comes from a genome so that the original genome sequence will be reconstructed. Although, we have not seen the factors, that affect the complexity of that problem. The two major factors are

- the number of the reads
- the length of the reads

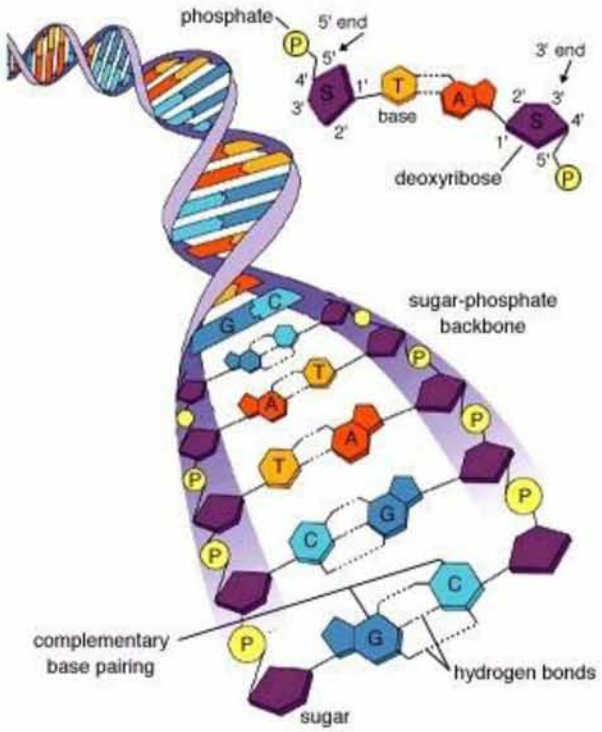
The dilemma with the number and the length of the reads is that, with more reads that are also longer is achieved better identification of sequence overlaps but the complexity behaviour is quadraticaly or even exponentially analogous to the number of the reads and the length of them. These values depend on the sequencing platforms.

Before the analysis of the sequencing platforms and methods let's take a closer look

at the reads and the structure of them. These fragments of DNA as mentioned previously are consisted from nucleotides. The generic molecular structure of the nucleotides is the following:



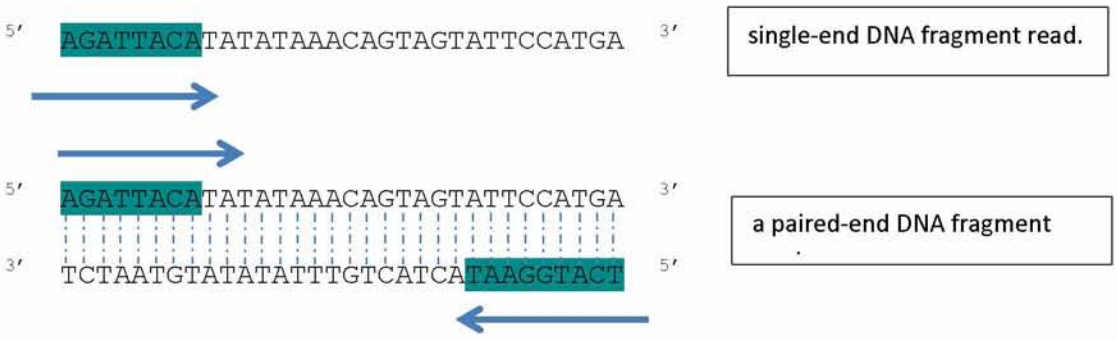
As shown the carbon atoms are numerated from the right to the left, 1 up to 5. These numbers are not random, the bond of the nucleotides on the DNA in order to form a strand, is always created between the ends with the numbers 3' and 5', through the phosphate group (P), the sugar-based backbone composing a polymer chain and a series of bases (B) extend above it. During the sequencing process the DNA is always extended with the 5th carbon with the direction to the 3' end, so that the DNA sequence is always read from its 5' end towards to the 3' end.



The double-stranded nature of DNA consisting of A, C, T and G nucleotides.

The sequencing process of a DNA fragment can be made in two different ways:

- sequencing only one strand of DNA from the 5' end, called the single-end (SE) reads
- sequencing the fragment in both strands from their 5' ends, called the paired-end (PE) reads.



At paired end reads another thing that is important is the orientation of the reads. The orientation of the reads depends on the sequencing technology. Each type of orientation has its significance.

Left Right, or (+/-), or Forward Reverse

This is the orientation of normal reads. The two reads are left and right respectively of the unsequenced part of the sequenced DNA fragment when aligned back to the reference genome.

LL,RR / (+/+) (-/-) / FF RR

Implies inversion in the sequenced DNA with respect to reference.

or

RL / (-/+) / RF

Implies duplication or translocation with respect to reference.

Another categorization of the assemblers can be done by the data reads are given as an input. These data are generated by the sequencing platforms. The sequencing platforms are distinguished in two categories based on the technology they use:

3.2.2 Categorizations and Analysis of Sequencing Technologies

The first generation ("Sanger") sequencing. Is the primar tool for the last 30 years, which is still widely used for small-scale experiments. It is based on the chain termination method using capillary electrophoresis. It is significantly slow compared with the new generation sequencers and expensive. Moreover, this process of sequencing can deliver read lengths up to 1000 bases, high raw accuracy, and allow for 384 samples to be sequenced in parallel, generating 24 bases per instrument second. Great shortcomings of this method are its high price, and long sequencing time. Examples of machines used for first generation sequencing include the Applied Biosystems Automated 3730 DNA Analyzer. To process a billion bases of DNA fragment it could take over a year to read and the cost would be excessive.

The new generation sequencing (NGS) methods. Compared to the Sanger method they generate reads much shorter and the total number of reads is respectively greater. The NGS methods where introduced in 2005 and lead the sequence machines to be able to go from serial to parallel and multiple sequencing production of DNA fragments simultaneously. The massive parallelism of NGS technology has greatly improved the throughput and lowered the cost of sequencing a genome. Among the most known NGS platforms we can name the Genome Sequencer from Roche 454 Life Sciences, the Solexa Genome Analyzer from Illumina, the SOLiD System from Applied Biosystems, Ion Storm and Ion Proton from LifeTechnologies, the Heliscope from

Helicos. There are several different methods that these technologies use for sequencing: parallelized pyrosequencing (454), sequencing by synthesis – cyclic reverse termination (Solexa), ligating degenerated probes (SOLiD), ion semiconductor (Ion Proton) and single molecule sequencing (Heliscope). Read lengths obtained from the NGS reads are in the 700bp range (454), the 100bp range (Solexa), the 75bp range (SOLiD) or less.

3.2.3 Comparison

To the next tables is made a comparison and gathering of characteristics between the most used new generation sequencing technologies. Also in the table exist and a Sanger machine in order to be seen the amount of difference between second and first generation.

Technology	**454 Sequencing**	**Illumina**	**Sanger**
Machine	GS FLX Titanium XL+	HiSeq 2000	3730xl
Sequencing Chemistry	Pyrosequencing	sequencing by synthesis (SBS)	Dideoxy chain termination
Mb per run	700 Mb	600 Gb (2 × 100 bp)	1.9~84Kb
Time per run	~23 hours	~11 days	20Mins~3Hours
Read length	700 bp (up to 1000bp)	~100bp	400~900bp
Cost per Mb	$ 84.39	$ 0.03	$2400
Accuracy	99.997% (15x coverage)	98%,(100PE)	99.999%
Cost per instrument	$ 500,000	$ 690,000	$ 95,000

Technology	**SOLiD**	**Ion Torrent**	**HeliScope**
machine	5500 series	Ion Proton™ Sequencer	tSMS™
Sequencing Chemistry	Ligation-based sequencing	Ion semiconductor sequencing	single molecule sequencing

Mb per run	170Gb	Up to 10 Gb	21 to 35 Gigabases per run
Time per run	~9 days	~2 hours (100bp)	~7 days
Read length	35bp + 75 bp	Up to 200 bp	25 + 55 bp
Cost per Mb	$ 0.04	$ 4,85	$ 0,005 per base
accuracy	Up to 99.99%	99.6%	99.995%
Cost per instrument	$ 595,000	$ 149,000	$ 999.000

instrument	**Advantages**	**Disadvantages**
Sanger	1.High quality 2.Long read length	1.High cost 2.Low throughput
Roche 454	1.Handles GC-rich regions (fairly) well 2.Long reads 3.Fast sequencing runs	1.Low throughput 2.Error rate at 1% 3.Homopolymeric regions are problematic
Solexa/Illumina	1.Highest throughput (max 600 gigabases/run) 2.Low cost per base	1.Error rate at 1% - only substitutions 2.Problems with AT- and GC-rich regions 3.Long sequencing times (dependent on the read length
SOLiD	1.High throughput 2.Low cost per base 3.High accuracy when a reference genome is available	1.Few software working with “color space” 2.Problems with AT- and GC-rich regions 3.Long sequencing times (dependent on the read length)
Ion Torrent	1.Fastest throughput 2.Inexpensive instrument	1.Relatively low throughput, hence high cost per base

	and consumables	2.Still prone to homopolymer error, though better performance than 454
HeliScope	1.Ease of library preparation	1.High error rates compared with other reversible terminator chemistries

The second-generation platforms give different reads of different size. It depends on the needs of the research which reads are more suitable for use. It is subject to the nature of the problem, if the researcher need long reads, or faster results, or the most accurate. Currently the Illumina platform is the most widely-used instrument.

It is obvious, that the genome centres adopted the new technology because of the much higher throughput and the lower cost. That had as a result the sequence assemblers to evolve in order to follow the new type of sequences. Primarily the assemblers that were created for long reads could also function for the sort reads, because the principals of detecting overlaps and building contigs are the same. However, this attempt failed due to poor performance and a variety of reason such as that many assemblers in order to work properly needed a minimum read length. Also the total number of reads is massive comparing to the previous technology. Another reason is that the algorithms based on long reads require a minimum amount of overlap that with short reads is impossible, because is too long.

In any sequence assembler the step that the overlaps are calculated is the most critical in order to have a good base to start. New short read sequencing projects had to be designed so that this step could be computationally realistic, and that is because, in order to have the same level of coverage that is achieved with the long reads are needed many more short reads. By coverage, is the average number of reads representing a given nucleotide, and so, 8X coverage indicate that the genome is sequenced eight times over. Considering this, with an increased amount of reads the computation of overlaps enlarges significantly. This problem is exacerbated by the fact that short-read projects compensate for read length by obtaining deeper coverage, and it is not unusual to see NGS projects at 30X, 40X, or 50X coverage rather than the 8X coverage that is typical of Sanger sequencing projects.

The assemblers that use the new generation sequencing platforms, had to approach differently the problem, set carefully the parameters used for computing the

overlaps and use new strategies and algorithms that work better with NGS data. The DNA sequencing data from the NGS platforms present shorter read lengths, lighter coverage and different error profile than Sanger sequencing data. The second-generation machines are characterized by highly parallel operation, higher yield, simpler operation, much lower cost per read, and shorter reads.

These are the main reasons that forced the creation of a new generation of genome assemblers, especially for handling the challenges of assembling very short reads. These assemblers include: Velvet, ALLPATHSLG, ABySS, and Ray. The common part of all these assemblers is that are graph based. But moreover are using the same method for building the graph, which at the moment is considered as the best method that copes with the illumina/Solexa reads (which are from the most widely-used instrument as previously mentioned) but also with a mix of reads of the sequencing technologies.

At the following chapter is made a comparison of the different graph methods that assemblers use and we conclude at the referring one.

3.3 Categorization of Assemblers based on graphs

The Next Generation Sequence (NGS), or Second Generation Sequence (SGS), assemblers can be organized into three categories, all based on graphs.

- The de Bruijn Graph (DBG) methods use some form of K-mer graph.
- The Overlap/Layout/Consensus (OLC) methods rely on an overlap graph.
- The greedy graph algorithms that can use OLC and DBG.

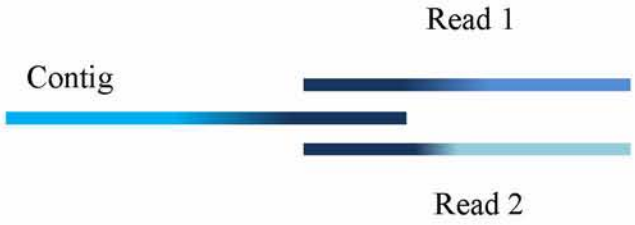
3.3.1 The Greedy Assemblers

They are the first of the NGS assemblers following a greedy strategy. They are checking, which are the reads that are most similar to each other and they join them.

The basic operation that characterizes the followed strategy is, given an unassembled read extent by adding one more read or contig that overlaps. Scanning through the unassembled reads the assemble process continues iteratively. Each time the next join is made by comparing all possible overlaps between the reads and assigning a score. Considering the highest- scoring overlap that it could be achieved, the reads are merged together. The process is repeated until no more extension are possible.

The greedy algorithms are considered as implicit graph algorithms, that they simplify the graph by passing only from the high-scoring edges. Also in order to avoid misassemblies the greedy algorithms have mechanisms that are terminating the extension process when conflicting information is found. The conflicting information is overlaps that two or more reads have with a contig, but the reads do not overlap each

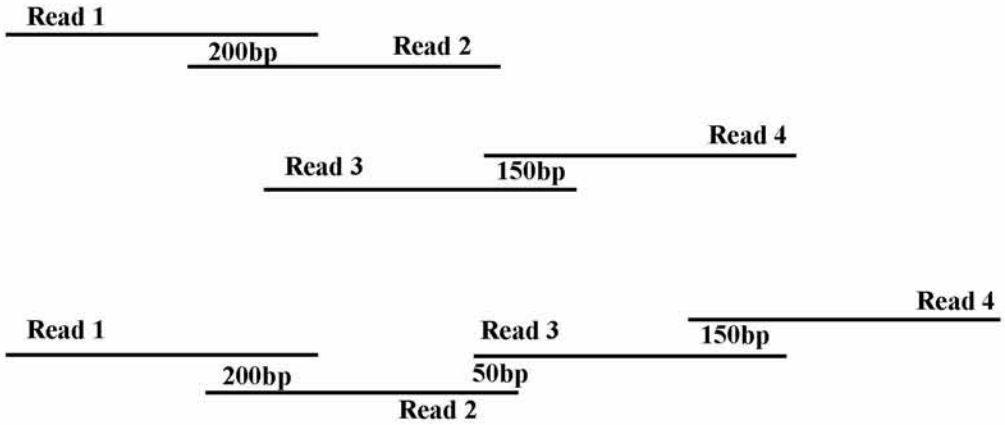
other. In this order, the contig would be extended false.



The dark blue area is where the two reads overlap with the contig, but the two of them do not overlap between them in the continue of their length.

However, although the greedy algorithms are easy at implementation, present dis-advantages. Because they consider the local information at every step and ignore the long range connexions between reads, the assembler could be easily confused by complex repeats because it is not easy to detect and resolve them. Also, all the current versions of assemblers that implement greedy algorithm are more memory intensive than the other implementations.

An example is shown below, where the assembler joins, in order, reads 1 and 2 (overlap = 200 bp), then reads 3 and 4 (overlap = 150 bp), then reads 2 and 3 (overlap = 50 bp) thereby creating a single contig from the four reads provided in the input. The disadvantage of the simple greedy approach is that because local information is considered at each step, the assembler can be easily confused by complex repeats, leading to misassemblies.



Some of the most known Greedy assemblers:

SSAKE, SHARCGS, VCAKE, QSRA

3.3.2 The Overlap/Layout/Consensus Assemblers

Currently the OLC approach is one of the two mainstream approaches of the de novo DNA assembly, and is especially used during the assembly of long (longer in comparison with the De Bruijn), high-quality reads. OLC assemblers use an overlap graph. Their operation has three phases in which first is finding all overlapping reads, next merge optimal overlapping reads into contigs and last derive the DNA sequence and correct read errors.

At the overlap phase each read is compared with every other read. This comparison in a pairwise manner is done in order to be constructed an overlap graph. All k-mers are sorted in reads and then the graph is builded by finding pairs of reads that are sharing a k-mer. Each read is a node and if an overlap exist between two reads then the nodes of these reads are connected with and edge. But these edges are constructed based on some criteria which differ among the OLC algorithms. Generally an edge is created if an overlap is at least K base pairs of length with least X% similarity. Also in this step

At the layout stage, the ultimate goal is to conclude in a single path that passes from each node in the overlap graph exactly one time. Graph algorithms are applied to the overlap graph that simplify and analyse him in order to identify the appropriate paths that correspond to segments of the genome sequence. These paths are collection of reads that overlap and are forming the contigs. The contigs in this case are sub graphs with many connections between the nodes that they contain. The sub graphs are compressed into one node so that the graph is simplified. Analysing these sub graphs is produced an approximate layout of the reads along the genome.

Finally at the consensus stage, the graph is reduced in large scaffolds by multiple sequence alignments. Ideally the original could be fragmented and composed from multiple scaffolds that have gaps between them. Anyhow, the goal is the original genome sequence to be inferred through the aligned reads.

3.3.2.1 Advantages-Disadvantages

Analysing the OLC algorithm approach, an advantage is that because the OLC assemblers are read based, the overlaps among the reads may vary in length. According to this, the input data of OLC assemblers could either from NGS or Sanger platforms.

Another advantage is that the three distinguished stages of the OLC approach

allow easiest optimization. The OLC algorithms, because of the natural three stage implementation, can be easily divided and modified to the needs of the assembly problem so that an optimization can be achieved.

However, the OLC approach is memory intensive and has a high processing cost. The overlap stage is very time consuming because of the determination of every overlap among every read of the sequence. In particular, the OLC approach was initially designed to receive input data from Sanger based platforms. These data are commonly a lot less than the data that are generated from the NGS platforms. The OLC assemblers can handle this kind of data, but because of the greater amount of data and the nature of them (short reads from NGS), can drive into two problems. The calculation time for the overlaps is increased significantly and too many ambiguous connections could be produced.

In addition, a very important observation is that in the layout stage, finding a path that traverses every node in the graph only once leads to a Hamilton path problem (the Travelling Salesman Problem), which is NP-hard and is not solvable in polynomial time. Because of this, OLC becomes more difficult and dependant on various heuristics to produce reliable results.

Some of the most known OLC assemblers:

Newbler, Edena, CABOG, Shorty, Forge

3.3.3 The de Bruijn Graph Approach assemblers

The algorithms that belong in this category have the following structure and every base-step is personalized in a different way in every algorithm. The basic structure of the algorithms relies on k-mer graphs, whose attributes make it suitable for vast quantities of short reads such these that are extracted of the NGS platforms.

3.3.3.1 De Bruijn Graph Construction (and Compression)

The common first stage of de Bruijn graph based de novo assemblers is to build the graph to store all the k-mers and their neighbouring nucleotides. This first stage can be summarized in following three steps; firstly the construction and the definition of all the unique length-k-mers. Next is done the node creation based on the k-mers and finally is done the edge creation based on the overlaps of the k-mers. The node and the edge construction in general is the same but personalize on the different algorithms. Also, when a k-mer is overlapping with multiple, different k-mers by k-1 length on the

side then multiple edges derive from the node that is dedicated to this k-mer. After the graph construction it is possible many of the assemblers to simplify and compress the graph without any loss of information.

3.3.3.2 Error Correction

Errors in assembly are found and corrected or removed once the graph is created. In order to move to the next step, the graph has to be cleaned from the vertices and the edges that are created at the graph construction step. Assembly errors can occur for two main reasons, because incomplete or incorrect information is provided to the assembler or due to the limitations of the assembly algorithm. Because of this, pieces could be incorrectly discarded as mistakes or repeats, and others could be joined up in the wrong places or with wrong orientation. In theory, the size of the de Bruijn graph depends only on the size of the genome and should be independent of the number of reads since it relies on k-mers. Though, because sequencing errors create their own graph nodes, inevitably the size of the de Bruijn graph is increased. Later will be explained the different kinds of errors and following with the algorithmic explanation of the assemblers will be seen how they handle them.

3.3.3.3 Scaffolding

At the step of scaffolding the algorithms analyze higher order information to achieve resolving ambiguities. This information is used to group, orient and join the continuous sequences that are formed, known as contigs, in order to link them to an unambiguous path in the de bruijn graph. This information could be paired-end sequences that are reads from the two ends of the DNA sequence and sequenced on GS FLX from the Roche 454 machine family. Mate paired sequences, that are reads similar to the paired end although are adapted to the Illumina HiSeq 2000 technology. Restriction map, that is a map of know locations of the DNA containing specific sequences of nucleotides. Or finally, clone maps or physical maps, that contain positional information about the contigs.

3.3.3.4 Finishing

Assemblies containing gaps are called draft assemblies, while the process of filling the gaps is called finishing. Finishing involves obtaining missing sequences, improving low quality regions, resolving misassemblies and ordering scaffolds. In order to fill gaps, further sequencing is usually required, either by amplifying and sequencing fragments spanning gaps, or by re sequencing the entire DNA. Tools that perform gap closure and finishing do exist although some hybrid assemblers perform gap closure indirectly by incorporating reads from multiple sequencing platforms such as Velvet.

3.3.3.5 The advantages of De Brujin Graphs:

The de bruijn graph based assemblers, conversely with the OLC assemblers which are read based, are k-mer centric. This is the fundamental difference between these two structures, while the OLC assemblers are depended directly from the length and the placement of the fragments, whereas the de bruijn assemblers are not affected how the reads are fragmented since they are k –mer centric. This is especially useful when attempting mixed-length sequencing or comparative genomics.

Also the use of de bruijn graphs give the assemblers the ability to reduce the memory consumption by having k-mers instead of reads as nodes. Additionally, because of the one to one relationship between paths and sequences that is formed due to the de bruin graph construction, the overlapping sequences follow necessarily the same path.

The de bruijn graphs have been regarded as heavy structures which require a lot of physical memory, but also because of the structure can efficiently be distributed on a cluster or small commodity computers . So, many parallel assembly tools based on the de bruijn approach exist, others because are especially designed for this purpose and other intergraded multithreading support in later versions.

Moreover, the error correction within the sequences it is significantly easier task using de bruijn graphs. The error correction is a major problem during the assembly process especially when using short reads. Also, the repeats are easier recognizable than in an overlap graph. (Although, the de bruijn graphs are sensitive to sequencing errors and repeats, because they lead to new k-mers and this adds complexity to the graph).

Finally, unlike the case of the OLC assemblers that have to find overlaps pairwise, the assemblers that use the de bruijn approach do not have to do all these explicit computations which is a very expensive computational process. For this reason, the de bruijn approach provides great performance when very large data sets are given. Finding an Eulerian path (de bruijn) through the edges of a graph is known to be a polynomial problem, and therefore can be executed more efficiently than finding a Hamiltonian path (OLC) []. Athough, because there can be many Eulerian paths in a graph, each de bruijn based assembler has to set constraints so that the path that represents the original genomic sequence can be found. The adding of this kind of constaints makes the assembly process more difficult, and could transform this polynomial problem to a NP-hard problem, so they have to be added carefully.

The disadvantages of De Brujin Graphs:

So far it is said that de Bruijn graph assemblers are much faster and can accommodate sequences of very different length. Although the de bruijn assemblers, as described above ,can handle both Sanger and NGS data, by dividing long reads from Sanger sequencing into short k-mers, there is an effective loss of long range connectivity information implied by each read.

Also, the de bruijn assemblers don't integrate all the calculations for finding pairwise overlaps to produce the overlap graph, because is no needed when following

this approach. But the disadvantage is that, especially with long reads, global pairwise alignments can be extremely useful to determine reliably if two reads really come from the same genomic locus. Constrained by the k-mer length, the de Bruijn graph can accumulate false-positive overlaps which could have easily been detected by pairwise alignment. These false-positive overlaps can make the resolution of repeats even more complex.

Some of the most known de bruijn assemblers:

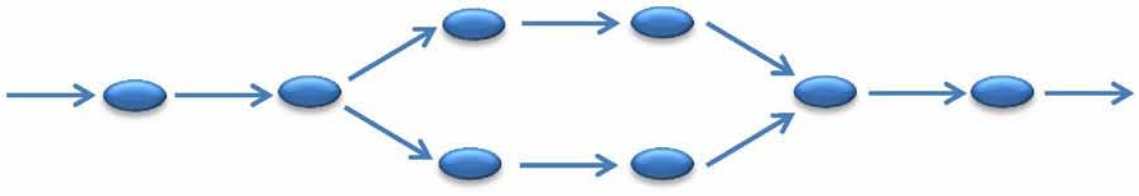
EulerSR, Velvet, ABySS, SOAPdenovo, ALLPATHSLG, Ray.

Considering the above comparison and based on the machine (which will be presented later) in which the genome assemblers are going to be examined, more suitable are the de novo de bruijn graph based assemblers that handle short reads.

3.4 Errors that occur during the genome asmebly process

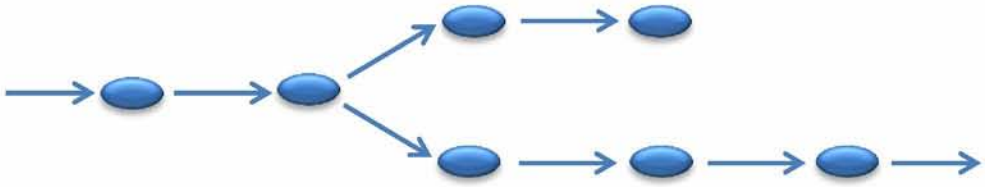
As it is said, sequencing errors make the de Bruijn graph more complicated as it is more sensitive to them, but many errors are easily recognized by their structure in the graph. Errors can bring the following problems de Bruijn graphs (but also in overlap graphs and generally, graph based algorithms):

The first kind of errors is the bubbles. Are errors in the middle of a read, that create alternate paths by separating a path into two branches that later join together forming again one path terminating at the same node. That kind of problems could be caused by sequencing errors that occur in the middle of a read. Also another reason could be the appearance of polymorphisms in the genome, and by polymorphism is meant when a genome contains two or more versions of the same sequence or set of sequences.

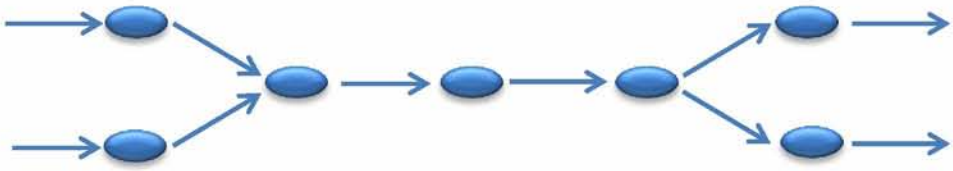


Continuing, the next category is the spurs, short dead-end branches (divergences) of the main path. This error can occur because the same pattern could be induced by coincidence of zero coverage after polymorphism near a repeat. Also a spur could be

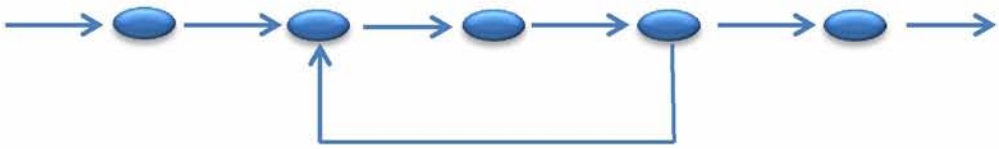
due to a sequencing error toward one end of a read.



The third category is converging and diverging paths. Known as the "frayed rope" pattern, is when two paths converge into on a path and afterwards diverges again into two separate different paths. The possible causes for this of error are repeats in the genome sequence.



And last, are the cycles. These errors are paths that at some point meet themselves and are caused of repeats in the genome.



Although sequencing technology has greatly improved, still no sequencing platform can produce sufficient data to assemble a complete genome from a single experiment. One of the key problems in shotgun sequencing is caused by repeats in genome sequences. Repeats, in cases when they are longer than fragment reads, induce problems during the overlap phase of assembly. As a result, assemblies result in fragmented contigs, separated by gaps. Repeat caused fragmentation is more pronounced in the NGS technology, since reads are generally of smaller size. Other than repeats, gaps can also be caused by other, technology-specific reasons.

4. Algorithmic analysis and description

Two assemblers are going to be tested and understand their behaviour ALLPATHSLG introduced on January 2011 by broad institute and Ray introduced by Boisvert S., Laviolette F., Corbeil J. on October 2010. They are assemblers based on de bruijn graphs but use different approaches. Both of them are expanding the graph using seeds, a method which will be explained later for both of the assemblers. The main difference is that ALLPATHSLG uses Shared Memory Parallelization and does not support distributed computing using MPI, so it will run in one machine. On the other hand, Ray is an assembler that can run in parallel on numerous interconnected computers using the message-passing interface standard. Moreover ALLPATHSLG handles the k-mer internally while Ray let the user choose the k-mer value. Below follows the analysis of the two approaches.

4.1 ALLPATHSLG

ALLPATHSLG is a whole-genome shotgun assembler that can generate high-quality genome assemblies using short reads (~100bp) such as those produced by the new generation of sequencers. The significant difference between ALLPATHSLG and traditional assemblers is that ALLPATHSLG assemblies are not necessarily linear, but instead are presented in the form of a graph. ALLPATHSLG uses the same steps as his predecessor ALLPATHS, which can routinely assemble small genomes, but has improvements so that can handle Large Genomes like mammalian.

The ALLPATHSLG algorithm starts with 2 pre-processor steps. The first is about read correction, and by this the algorithm applies a step in which identifies the trusted K-mers. A K-mer is concidered as trusted when it occurs at high frequency in reads and at high quality. Each K-mer is examined and the algorithm tries to find all of its instances in the reads, then the collection of the reads is examined for the quality score for each base in the K-mer. The algorithm retains reads whose K-mers are trusted. Untrusted K-mers can be rehabilitated from the algorithm and change their status to trusted. This could be under two conditions, in the first one reads are restored if there are up to two substitutions to low quality score calls, which makes its K-mers trusted. An in the second one, K-mers are restored later if they are essential for building a path between paired-end reads.

Continuing, at the second pre-processor step, the algorithm creates unipaths. At this step the algorithm uses the reads to compute the unipaths. It finds the (K+1)-kmers in the reads whose first and last Kmers are trusted .The unipaths are mathematically defined by the trusted kmers together with these (K+1)-mers, which define adjacencies between the trusted k-mers. The algoritm continues, with the calculation of perfect read overlaps seeded by these K-mers. Every K-mer gets a numerical identifier corresponding to the appearance frequency of it in reads and overlaps. The algorithm then merges intervals to an extent that is consistend with all the reads. The operation is equivalent to de Bruijn graph construction. The database implementation reduces the RAM requirement for the graph construction, which comes next. Also at this step the

algorithm tries to group together most or all the reads of a given region of the genome, which aims to help the algorithm later to tile the genome by overlapping regions, assemble each region separately and then glue them together to form one big assembly of the entire genome.

ALLPATHSLG now builds a de bruijn graph from the database implemented in the previous step. The first graph operation is spur erosion, which it calls unipath graph shaving. After the initial creation of the unipaths , in many cases read errors result in short terminal branches (spurs) within the graph. Those that the length of them is shorter than 20 K-mers, are removed in order to make clearer that there is a longer alternative branch. Moreover, ALLPATHS apply a unipath recovery step, At this step the algorithm identifies unipaths that are not represented in the current structure of assembly. It tries to extend them unambiguously where possible, with the aim to add them to the assembly. The point of this step is to find small regions that have relatively high copy number. Copy number is a number which infers from read coverage of the unipaths.

Having set the bases with the help of the previous steps, at next ALLPATHSLG partitions the graph that as said above aims to resolve genomic repeats by assembling regions that are locally non-repetitive. At first, the algorithm chooses the seeds. Seeds are called the unipaths around of which the algorithm is going to build these regions. Seeds that are long and have low copy number are ideal, but is not necessary to use every ideal unipath as a seed. The choosing of seeds is done by starting from a set of all the unipaths and iteratively remove the non-suitables until no unipaths can be removed from the set. The evaluation process for each unipath starts by finding its left and right neighbour unipaths using short-range paired-ends (for a tighter variance). If the distance between the neighbours is less than a threshold the given evaluated unipath is removed.

The next step is the neighbourhood building around the seeds. The goal is to assemble the extensions around the two sides of the seeds. As neighbourhood is considered the genomic region that contains the read and extends 10 kb on each one of the two sides of it. This is achieved using a set of low copy unipaths and two sets of reads. The unipaths that are used are those covering partially the neighbourhood. The first set of reads contains those that concur with the neighbour unipaths, and the partners of these reads. In the second set are contained all the short-fragment read pairs that the sequence of the both pairs can be assembled from the first set reads. The second set may contain pairs outside of the neighbour region. The problem is that the second set is quite numerous, so the algorithm merges these short-fragment read pairs, thereby taking a decreased set with more information and less closures.

Then, the algorithm performs a step that glues together the closures of the local assembly forming in this way the neighbourhood graphs. This is analogous to joining contigs based on sequence overlaps. This step works by iteratively joining closures that have long end-to-end overlapping stretches. . But in many cases the process will glue together some identical sequences that come from different parts of the genome, forming long perfect repeats. ALLPATHSLG assembles each partition separately and in

parallel, end when the process finishes the algorithm glue all of these local assembly graphs in a single sequence graph. The graph could have several components; this is analogous to the number of the chromosomes and the success of the process. The next move of the algorithm is to handle the above mentioned perfect repeats, while building the global assembly graph, by assemble them on top of each other. These collapsed repeats of the assembly may be encountered at the next step which is the editing part, where this imperfect representation of the genome sequence is improved with heuristic methods. Allpaths remove spurs, small disconnected components, paths not spanned by paired-ends and It also uses paired-ends to tease apart, where is possible, collapsed repeats that display the frayed rope pattern. Consistent it is.

This is the basic structure that ALLPATHSLG follow to assemble. ALLPATHSLG in order to handle larger genomes had some improvements by the time such as the use of jumping libraries, which is helpful due to the fact that the junction point of a jump will often lie within one of the two reads. Moreover, the data sets of mammalian genomes are large (~3 × 109 reads), and the handling of overlapping reads in this size of genome require a large amount of data be held concurrently in memory. Due to this, the algorithm uses more economically the data structures and makes efficient use of shared memory parallelization.

4.2 Ray

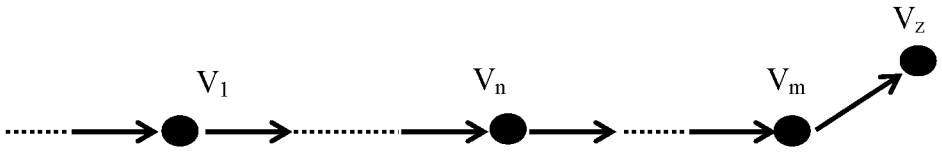
The Ray algorithm is an implementation of greedy algorithm on a de bruijn graph. Is a parallel short-read assembler and has been developed to distribute the graph across multiple computers through MPI. Also allows simultaneous assembly of reads from a mix of new generation sequencing technologies.

Ray is based on a de bruijn graph and follows the next steps for the genome assembly problem. At first the algorithm builds the de bruijn graph that is defined as follows, given a family of reads D, the de bruijn parameter k, which stands for the size of k-mer, and a coverage cut-off c. The V vertices of the graph are k-mers that are c-confident, which means that each k-mer that participates in the graph appears at least c times in the reads D. And the set of the edges is composed of all the edges between two V vertices that form a (k+1)-mer with l-confidence. In example for k=4 and c=2, in order to be vertices of the graph the k-mers ATGT and TGTC , have to appear at least 2 times in the family of reads D. Also the edge between these two k-mers will be in the graph only if ATGTC appears at least once in the family of reads D.

Moreover, in order to reduce the running time and the memory consumption is applied a simplification step to the graph. At this step, the sequence data that are provided by the reads are used through a de bruijn graph annotation, and each read is being annotated only once and attached to a single vertex of the graph. Since each read is annotated only once, this step is not memory intensive.

Furthermore, Ray does not perform error correction like the most de Bruijn based assemblers. Instead, it just avoids erroneous k-mers. The user has to be careful with a k-mer that is too large because Ray does not attempt at all to correct the reads. The erroneous k-mers all go in an "abyss" and are not really utilised at all.

During the next step, the algorithm defines specific sub sequences that are called seeds. The seeds are paths that are strongly considered that are sub sequences of the genomic sequence, based on the criteria stated by the algorithm. This is achieved by considering a de bruijn graph with a great cut-off value. The cut-off value is defined as the average of the average coverage and the number that indicates where the number of errors is lower than the number of true sequences. This process gives a de Bruijn graph that has a very small amount of errors but also, this stringent graph has a huge loss of information contained in the reads. This loss of information does not really affect the algorithm because this process is only determining the seeds and not the form of the de bruijn graph. The above procedure gives in the graph only the vertices that participate in the seeds. While the algorithm defines and chooses these seeds, the process is followed by the extension of each one of these seeds into a contig. This process is controlled by heuristics that are applied during the extension process. The process of an extension is stopped if at some point the family of reads does not clearly indicate the direction of the extension. For example, if we consider the following graph, and say that the seed is consisted of the v_n until the v_m , then the v_z will be added if most of the reads that overlap the seed are also containing the v_z. If this does not happen the algorithm chooses another direction. But, in the case that there is no obvious extension the process stops.



According to the criteria of the heuristics, the reads that overlap numerous times a contig have higher importance in contrast with the reads that traverse only a small number of the last vertices of the contig. This is because, according to the philosophy of the algorithm, the reads which overlap at a greater degree with the given contig that is constructed, are more suitable and contain better information about the direction in which the extension should proceed. The overlapping degree is measured with the functions offset and offset ^{paired} . With this process and applying the specific heuristics the length of the contigs is limited when compared with other algorithms contigs but

then again give the advantage of decreasing the assembly errors. Furthermore, by the use of these heuristics the algorithm obtain a greedy traversal strategy. This means that Ray uses greedy algorithms for choosing which extension should follow in order to have the desired structure. The greedy choice is considered so that the algorithm runs in polynomial time.

This is the logic that Ray follows in order to expand from seeds to contigs and then to scaffolds so that in the end the output is a completed genomic sequence ideally or a group of contigs and scaffolds with a respective high number of coverage.

5. Hardware and Software inquiry of Magerit

5.1 Profile of the machine

Magerit is a cluster consisting of 260 computer nodes, of which 245 nodes are eServer BladeCenter PS702 2S with 16 Power7 processors 3'3 GHz (26.4 GFlops) and 32 GB RAM, and the rest are 15 nodes eServer BladeCenter HS22 with eight Intel Xeon 2'5GHz (10.2 GFlops) processors with 96 GB RAM, implying 4,160 CPUs and 9.2TB RAM. All the nodes operate independently and with the same software configuration. The system has a local storage capacity of about 192 TB, provided by 256 disks of 750 GB, which uses a distributed and fault tolerant system (GPFS).

Because of its size, the system is configured to process packets of jobs in batch mode. It uses the SLURM queue manager that plans various jobs with the dual aim of maximizing the use of computer power and process jobs of different users as quickly as possible.

5.2 SLURM (Simple Linux Utility for Resource Management)

SLURM is an open source, highly scalable and fault-tolerant cluster manager and job scheduling system for large clusters of compute nodes. SLURM provides three key functions, the first is that allocates in an exclusive and/or nonexclusive fashion the access to resources ,which are the computer nodes, to the users for a duration of time so they can perform a job. Also, distributes jobs to a set of allocated nodes with a framework for starting, executing, and monitoring job, which typically is a parallel job (such as MPI). Finally, maintain a queue of pending jobs and manages the overall utilization of resources.

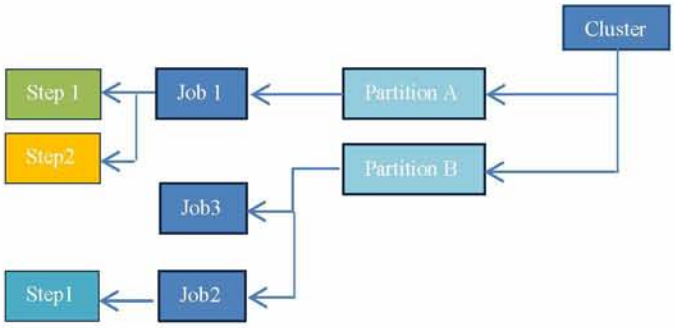
The configuration of the queuing system aims to ensure that the users have fair

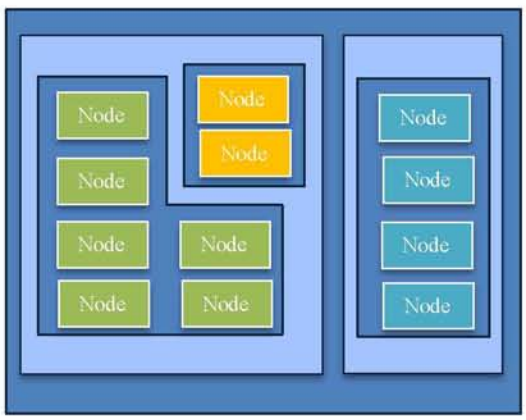
access to the resources, and that them are shared fairly. The approach that is used is known as fairshare scheduling. Whether a job is started before another job depends on two factors, when the job was submitted and how many resources are available to the user. In general, jobs belonging to users who, in the recent past, have not consumed much CPU time will tend to start before jobs that belong to more active users (by the meaning that have used more CPU time in the resent past). But, this mechanism is essential only when jobs compete for resources. If enough resources are available, a job will be started.

The architecture of SLURM can be characterized as a fairly traditional cluster management. At the top is a pair of cluster controllers, which work as the managers of the compute cluster, implementing a management daemon, called slurmctld. The slurmctld daemon monitors the resources, but the most important part of its work is that maps incoming jobs to the compute resources.

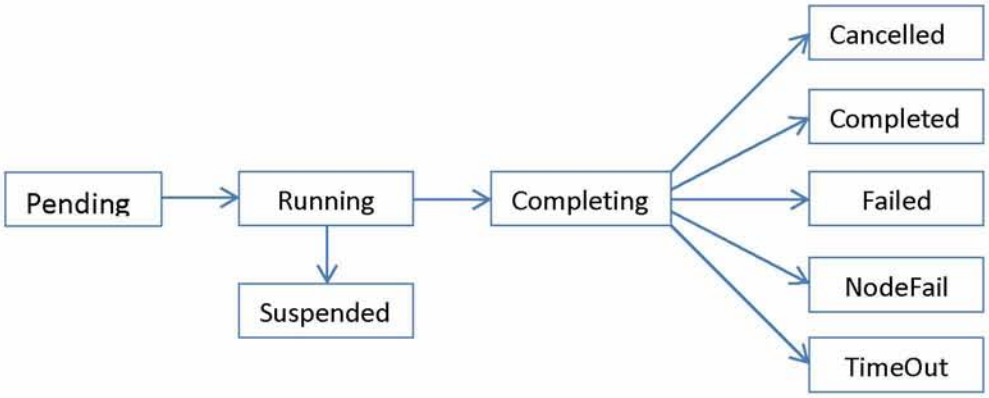
Each node implements a daemon called slurmd. The job of slurmd daemon is the management of the node and the monitoring of the tasks that are running on the node. Also, slurmd daemon, accepts jobs from the controller, and maps these jobs to tasks on the cores within the node. Moreover, if it is requested from the controller, it can stop tasks for executing.

Furthermore, other daemons exist but these are the most important that concern our work. The SLURM entity is consisted of different partitions. These partitions are sets of nodes, which are individual computers, collected in logical groups. The partitions can be configured with constrains about the time size limit of the jobs or about the users that allowed to use them. Also, commonly the partitions include a queue of incoming jobs. In more detail, a job is a resource allocation, a refinement of the partition that is mapping a set of nodes inside the partition for a user for a certain period of time for work. Each job consists of one or more job steps. The job steps are sets of (typically parallel) tasks that are executed on that prescribed subset of nodes.





The jobs, as explained above, are executing resource allocation, which means specific processors and memory or entire nodes allocated to a user for a given time period. The jobs can be interactive and executed in real-time or a script queued for later execution, called batch. The user is allowed to set constrains that are available to the execution of the job and each job is distinguished by an ID number identification. The states that a job can be in the system are the following:



6. Use of assemblers and Building

6.1 ALLPATHSLG

6.1.1 ALLPATHS LG How to Build

6.1.1.1 Requirements

To compile and run ALLPATHS-LG you will need a Linux/UNIX system with at least 16 GB of RAM. The suggested are, a minimum of 32 Gb for small genomes, and 512 Gb for mammalian sized genomes

6.1.1.2 System Requirements

The assembler has been builded and tested by the Broad Institute on a modern version of Linux for the x86_64 architecture. ALLPATHSLG does not run on 32-bit machines, it is necessary to have a 64-bit Linux system. It can be builded and executed in a variety of Linux distributions including Ubuntu, RedHat, and SUSE. It can work fine in any distribution of x86_64 Linux, as long as the system provides the basic compiler and library requirements. These are the next packages:

GCC, with its associated g++ compiler for the C++ language.

The GMP library compiled with the C++ interface, the GCC installation may already include GMP

Also the program requires the architecture type of the machines processor to be little endian.

6.1.1.3 The Build Procedure

After downloading the package from the FTP site, move it to a location on the system where it is preffered the software to be builded. (Although be careful when downloading from the ftp server)

```
% tar xzf allpathslg-<release>.tar // Extract the contents
```

After extracting move into the source directory, using the command:

```
% cd allpathslg-<release>.tar
```

Then execute the configuration script by typing:

```
% ./configure --prefix=/path/to/install/directory
```

At this point for the specific intel node on which the program was builded, a change has been made in the configure file in order to accept the gcc compiler:

```
ax_compare_version=`echo "x$ax_compare_version_A

 x$ax_compare_version_B" | sed 's/^ *//' | sort -r | sed
"s/x${ax_compare_version_A}/true/;s/x${ax_compare_version_B
}/false/;1q"`
```

`ax_compare_version=true` (this is the added line)

And after build the software with:

```
% make
```

here the user can set the parameter `-j<n>`

`-j<n>`Split the compilation into n parallel processes. If n is set equal to the number of CPUs on the machine, it will speed up compilation approximately n-fold. It is a useful tip because the building of the code is a lengthy process.

Finally, install it with:

```
% make install
```

The executables are in `/path/to/install/directory/bin` but the user has to make sure this is on his PATH. Either with the export command every time is going to use the program or adding the path to the `.bash_profile file`.

6.1.1.4 ALLPATHS pipeline overview

ALLPATHS is consisted of a series of modules, where each module performs a step of the assembly process. These different modules is possible that the user can run them manually, indivisualy and in varying order depending on the parameters of the assembly. But alternatively, and as recomented from the manual, the user is able to run a single module called RunAllPathsLG that controls the entire pipeline, deciding which modules to run and how to run them. The RunAllPathsLG module, uses the Unix make utility to control the assembly pipeline and creates a special makefile that call each module.

6.1.1.5 Preparing data for ALLPATHS

The ALLPATHSLG uses a specific pipeline directory structure and before the assembling process the data should be prepared and imported. Moreover, the input data should meet certain requirements such as that any input dataset should include at least one fragment library and one jumping library. As fragment library is defined a library with a short insert separation. The insert separation should be less than twice the read length, so that the reads may overlap. For example reads from 100bp generated from 180bp inserts. As a jumping library, is defined a library with longer separation, which typically varies in a range from 3kbp to 10kbp. The libraries that are going to be used should have about 45x coverage.

Moreover ALLPATHS also supports long jumping libraries, this kind of libraries are optional and used only to improve the scaffolding process in mammalian-sized genomes. As a jumping library, is defined a library with an insert size that is larger than 20

kbp. In long jumping libraries the coverage is not required to be so high as the previous libraries, typically long jump coverage of less than 1x is sufficient to significantly improve scaffolding.

Any other type of library construction is not supported by ALLPATHS at this point.

Generally, the input format data is specified by the model seen in the next table.

Libraries, insert types	Fragment size	Read length	Sequence coverage	Required
Fragment	180bp	≥100b	X45	Yes
Short jump	3,000bp	≥100b preferable	X45	Yes
Long jump	6,000bp	≥100b preferable	X5	No
Fosmid jump	40,000bp	≥26b	X1	No

ALLPATHSLG team has adopted this model because applying it only a few libraries are required to be constructed. In this way the laboratory load of work is a lot less in manners of time and required DNA amount. Additionally, the fragment library has the characteristic that the inserts are ~1.8 times the sequencing read length. This ensures that the inserts are short enough so that sequencing reads from each end overlap by ~20% and can be merged to create a single longer "read". As a final point, by using "jumping libraries" ALLPATHSLG achieve to obtain long-range connectivity, since current technology cannot sequence fragments greater than ~1 kb.

Another point that the user of AllPATHS should be very careful and examine before using the libraries is the read orientation. The reads of the fragment libraries are expected to be oriented towards each other (inward).

On the other hand, the reads of the Jumping library are expected to be oriented away from each other (outward), as a result of the typical jumping library construction methods.

Finally, the reads of the long jumping libraries are expected to be oriented with the same orientation as the fragment libraries reads that is towards each other (inward).

Having specified and selected the input libraries that the algorithm is going to receive, the next step is to define the files and run the `PrepareAllPathsInputs.pl` script that is going to prepare the field for the assembly process.

An example of the `PrepareAllPathsInputs.pl` script running through a bash file appropriate for Magerit is given below,

6.1.1.6 Running conversion script

```
#!/bin/bash
#----------------------- Start job description ----------------------
#@ class                = standard
#@ partition            = intel
#@ initialdir           = /home/A10100002/visitor102/E_coli/preprun
#@ output               = /home/A10100002/visitor102/E_coli/preprun/errout/%j-out.log
#@ error                = /home/A10100002/visitor102/E_coli/preprun/errout/%j-err.log
#@ total_tasks          = 1
#@ tasks_per_node       = 8
#@ wall_clock_limit     = 06:00:00
#------------------------ End job description ------------------------

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

# Run our program
#ALLPATHS-LG needs 100 MB of stack space.  In 'csh' run 'limit stacksize 100000'.
ulimit -s 100000

# NOTE: The option GENOME_SIZE is OPTIONAL.
#       It is useful when combined with FRAG_COVERAGE and JUMP_COVERAGE
#       to downsample data sets.
#       By itself it enables the computation of coverage in the data sets
#       reported in the last table at the end of the preparation step.

PrepareAllPathsInputs.pl\
 DATA_DIR=/home/A10100002/visitor102/E_coli\
 PLOIDY=2\
 IN_GROUPS_CSV=in_groups_e.csv\
 IN_LIBS_CSV=in_libs_e.csv\
 OVERWRITE=True\
 | tee ecolipre.out

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

DATA_DIR

Is the directory that holds the information that is essential in order to do the assembly process. The data directory for ease of use has the name of the genome that is going to be assembled but is always in the choose of the user. This directory should contain files that have the sequenced reads, the quality scores of them, information about their pairing, and also the ploidy file. These files may already exist if the user continues or restarts an existing assembly.

PLOIDY

Ploidy is the number of sets of chromosomes in a biological cell. The file ploidy, contains the number that indicates the ploidy of the genome. Is a single line file contain-

ing the numbers 1 or 2. This number indicates the ploidy of the genome with 1 for haploid genomes and 2 for diploid genomes. Polyploid genomes are not currently supported. The specific file name is:

<REF>/<DATA>/ploidy

IN_GROUPS_CSV:
The in_groups.csv file contains the location of the libraries that are going to be used and has the following structure:

In_groups.csv

```
                    file_name,  group_name, library_name,
  /seq/Illumina/011/302.bam,        frags,  Illumina_011,
 /seq/Illumina/012/303.?.fasta,     jumps,  Illumina_012,
/seq/PacBio/007/100.*.fastq.gz,     short,   PacBio_007,
```

Each column in `in_groups.csv` file provides, for each data file, the following information:

`file_name`:
this field contains the complete path to the data file. Because the paired reads files are distinguished with 1 and 2 or A and B, or may be used multiple unpaired fastq or fasta files, the in_groups.csv file structure permit the user to make use of wildcards '*' and '?' but with the restriction that cannot be used in the extension. Extensions that are supported: '.bam', '.fasta', '.fa', '.fastq', '.fq', '.fastq.gz', and '.fq.gz', all case-insensitive. For '.fasta' and '.fa' it is expected that corresponding '.quala' and '.qa' files exist, respectively.

`library_name`:

it is field containing the name of the library to which the data set belongs. Usually is the platform which was used followed by an ID number. Additionally, has to be the same with the corresponding field at in_libs.csv file.

`group_name`:
a UNIQUE nickname for this specific data set.

IN_LIBS_CSV
The In_libs.csv file contains information of the different files to be converted, and has the structure that is shown next:

In_libs.csv

```
library_name,  project_name, organism_name, type, paired, frag_size, frag_stddev, insert_size, insert_stddev, read_orientation, genomic_start,
genomic_end
 Illumina_011,      test_h_1, human_Chromosome_14,   fragment,        1,        155,        50,              ,              ,        inward,
0,
 Illumina_011,      test_h_1, human_Chromosome_14,    jumping,        1,          ,         ,          2500,            500,       outward,
0,
   PacBio_007,      test_h_1, human_Chromosome_14,       long,        1,          ,         ,         35305,           1000,        inward,
0,
```

Each column in `in_libs.csv` describes a specific field of a library. These fields are:

`library_name`:
it contains the library name and matches the same field in `in_groups.csv`.

`project_name`:
it is a string that is characterizing and naming the project.

`organism_name`:
the organisms name and probably some identification number of the experiment or the sample.

`type`:
fragment, jumping, etc. This field is only informative.

`paired`: `0`: Unpaired reads

`1`: paired reads

`frag_size`:
contains the average number of bases in the fragments and is only defined for fragment libraries.

`frag_stddev`:
it is a number that is the estimated standard deviation of the fragments sizes and is only defined for fragment libraries.

`insert_size`:
contains the average number of bases in the inserts it is defined only for jumping libraries. If the insert length in higher than 20 kb then the library is considered to be a long jumping library.

`insert_stddev`:
it is a number that is the estimated standard deviation of the inserts sizes and is only defined for jumping libraries.

`read_orientation`:
can take only two values inward or outward. The outward oriented reads will be reversed.

`genomic_start`:
has the index of the first genomic base in the reads, if the value is non-zero then all the bases before `genomic_start` will be trimmed out.

`genomic_end`:
has the index of the last genomic base in the reads, if the value is non-zero then all the bases after `genomic_end` will be trimmed out.

After a successful run of PrepareAllPathsInputs.pl the necessary ALLPATHS input files should be in place and ready for an assembly run to start.

6.1.1.7 Running ALLPATHS

```
#!/bin/bash
 #---------------------- Start job description ----------------------
 #@ class                = standard
 #@ partition            = intel
 #@ initialdir           = /home/A10100002/visitor102/all_human_Chromosome_14/preprun
 #@ output               =
/home/A10100002/visitor102/all_human_Chromosome_14/preprun/errout/%j-out.log
 #@ error                =
/home/A10100002/visitor102/all_human_Chromosome_14/preprun/errout/%j-err.log
 #@ total_tasks          = 1
 #@ tasks_per_node       = 8
 #@ wall_clock_limit     = 06:00:00
 #---------------------- End job description ----------------------

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

 # Run our program

 RunAllPathsLG\
  PRE=/home/A10100002/visitor102\
  REFERENCE_NAME=all_human_Chromosome_14\
  DATA_SUBDIR=.\
  RUN=run\
  SUBDIR=test_h_8\
  THREADS= 8\
  OVERWRITE=True\
  | tee -a humanrun16.out

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

PRE

The command-line argument PRE specifies the location of the pipeline directory structure.

REFERENCE_NAME

The REFERENCE directory is the directory containing the reference genome that is going to be assembled. The use of this argument is done in order to separate the assembly projects by organism and also by isolate if the user has two different references of the same organism. The naming of the REFERENCE directory usually occurs of the organism. For each given organism or an isolate of it, the REFERENCE directory will contain all the assembly projects. In this directory will be stored all the

intermediated files that are going to be generated. The need of a reference genome is not required but is useful while the algorithm can perform evaluations at the several stages of the assembly process that are going to be useful on the duration. In any case, even with the absence of a reference genome, the user should create a REFERENCE directory for the organism that is going to be used. Moreover, in the REFERENCE directory the user could store various DATA directories each one for a certain set of read data to assemble.

DATA_SUBDIR

The DATA directory, as the name declares, contains the original read data that are used in a particular assembly process. The form of the data is in fastb, qualb, pairs which are internal ALLPATHS formats. In this file also are existent the intermediate files that are resulting from the original data. These files are used normally during the process of evaluation and are independent from the individual assembly tries. At each DATA directory the user could set various RUN directories each one for a particular attempt to assemble the original data using a different set of parameters.

RUN

In the RUN directory are located all the intermediated files that are generated during the preparation for the final assembly stage from the original data reads. In addition, this file may contain the intermediate files that are used in the evaluation process and are independent on the each particular assembly attempt and the parameters of them.

SUBDIR

At the SUBDIR directory the user can locate the files of the final assembly process. Moreover, there are some intermediate files produced during the assembly process and some evaluation files.

THREADS

The THREADS argument is declaring the number of threads that the user is going to make use of. The maximum number that makes sense is the number of the threads the node supports.

Kmer size, K

A sensible observation that can be made it that nowhere is defined the k-mer value. As suggested from the ALLPATHSLG team the user should not adjust the K-mer size from the default value of K=96. As previously mentioned the ALLPATHSLG uses k-mers of various sizes internally.

6.1.1.8 Assembly Results

The results of the assembly pipeline are given in the following two files in the assembly `sub_dir` directory:

`final.assembly.fasta and final.assembly.efasta`

Both these files contain the final flattened and scaffolded assembly. The `efasta`, "enhanced" `fasta`, file is a new format used by ALLPATHSLG and is based on the standard `fasta` file format.

The user may also find useful information and statistics about the process in the following files:

```
assembly.report
assembly_stats.report
final.summary
library_coverage.report
```

and information about some errors that could occur during the assembly pipeline in the file: :

```
%j-err.log
```

also useful information about the algorithms outputs and how the assembly pipeline developed the user can get them from the file:

```
%j-out.log
```

6.2 Ray

6.2.1 Requirements

It is required that the system in which Ray is going to be builded will have a C++ compiler, make and an implementation of MPI.

6.2.2 The Build Procedure

// Extract the contents

```
tar xjf Ray-x.y.z.tar.bz2
```

After extracting move into the source directory, using the command:

```
cd Ray-x.y.z
```

And after build the software with:

```
make PREFIX=build
```

here the program comes with some spesifications that the user can set according to the assembly, the user can change the compiler:

```
make PREFIX=build MPICXX=/software/openmpi-k.l.m/bin/mpic++
```

or the user can change the maximum length of the k-mers that are going to be used (the default maximum k-mer is 31) .Larger k-mers utilise more memory.

```
make PREFIX=build MAXKMERLENGTH=64
```

or can change both of them

and install it with:

```
make install
```

after the user can find and use the assembler which is located at the prefix folder :

```
ls build
```

6.2.3 Data preparation for Ray

The Ray does not use such a strict pipeline directory as the previous algorithm. The user does not have to make such a preparation of the data before the assembling process, although should meet some requirements.

The algorithm can function without the presence of jumping libraries, only using fragments libraries, though would be helpful. Anyhow, the reads of the libraries that are going to be inserted should have inwards or outwards orientation. The Ray cannot handle other libraries.

If the library that the user is going to import does not meet these requirements can use the FASTX-Toolkit, a short read preprocessing tool(appendix).

Ray does not require any specific coverage of the library.

The supported input files are the following:

.fasta
.fastq
.csfasta (color-space reads)
.sff (paired reads must be extracted manually)

Also other types of files are supported,basically the same but compressed, however the user should first define some parameters at the compilation step.

.fasta. gz	set the HAVE_LIBZ=y at compilation

.fasta.bz2	set the HAVE_LIBBZ2=y at compilation
.fastq.gz	set the HAVE_LIBZ=y at compilation
.fastq.bz2	set the HAVE_LIBBZ2=y at compilation

6.2.4 Run Ray

```
#!/bin/bash
#---------------------- Start job description ----------------------
#@ class                = standard
#@ initialdir           = /home/A10100002/visitor102/resultray/Human_14_outputs
#@ output               = /home/A10100002/visitor102/resultray/Human_14_outputs/errout/%j-
out.log
#@ error                = /home/A10100002/visitor102/resultray/Human_14_outputs/errout/%j-
err.log
#@ total_tasks          = 512
#@ tasks_per_node       = 16
#@ wall_clock_limit     = 20:10:00
#----------------------- End job description -----------------------

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

# Run our program
srun ~/Ray-v2.0.0-rc7/build/Ray -n 512 -k 45 \
 -p ~/Human_Chromosome_14/frag_1.fastq ~/Human_Chromosome_14/frag_2.fastq\
 -p ~/Human_Chromosome_14/shortjump_1.fastq ~/Human_Chromosome_14/shortjump_2.fastq\
 -p ~/Human_Chromosome_14/longjump_1.fastq ~/Human_Chromosome_14/longjump_2.fastq\
 -o Human51245

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

To run Ray the user can put the executable in the PATH or can call Ray by the full path, in both cases from the bash file.

-n

The `-n` parameter is the number of the ranks that the algorithm in going to use and must match the number of the total_tasks in the job description section.

-k

The -k parameter set the k-mer length, selects the length of k-mers that the algorithm is goingto use for the assembly process. The default value of the k-mer if the user do not set it, is 21. The value of the -k must be odd and cannot take even numbers because reverse-complement vertices are stored together.

The choice of the k-mer length depends on the needs of the given genome library and the characteristics that the user want to provide to the assembly. The choice of a k-mer length too small though could provoke many ambiguous k-mers. On the other hand, a choice of a k-mer length too large could cause sequencing errors and destroy the connectivity while also more memory is needed. Longer k-mer provides more specificity but also decreases the usable coverage. A practical recommendation is the user to scan on a certain range for the proper k-mer length by trying several values and then should pick the assembly with large N50 value and long contigs. practical recommendation: parameter scan - try several values for k and pick assembly with long contigs

6.2.5 Input files and their declaration

Ray handles the following input files with the declaration shown next:

`-p`
leftSequenceFile rightSequenceFile [averageOuterDistance standardDeviation]
The user provides two files containing paired-end reads, the user has the option not to provide the averageOuterDistance and standardDeviation values, then the algorithm automatically compute them.

`-i`
interleavedSequenceFile [averageOuterDistance standardDeviation]
The user provides one file containing interleaved paired-end reads, as above the user has the option not to provide the averageOuterDistance and standardDeviation values, then the algorithm compute them automatically.

`-s` sequenceFile
The user provides a file containing single-end reads.

`-o`
With the `-o` parameter the user specifies the directory that the algorithm is going to use for the output files. If not declared the default is RayOutput. If the user attempt to make another assembly with the same output file the algorithm is going to crash, so it is important at each assembly attempt the user to specify an output directory with different unique name.

6.2.6 Assembly results

The results of the Ray assembly process are the following:

Contigs:

- `Contigs.fasta`, contains the contiguous sequences in FASTA format
- `ContigLengths.txt`, contains the lengths of contiguous sequences

Scaffolds:

- `Scaffolds.fasta`, contains the scaffold sequences in FASTA format
- `ScaffoldComponents.txt`, contains the components of each scaffold
- `ScaffoldLengths.txt`, contains the length of each scaffold
- `ScaffoldLinks.txt`, contains the scaffold links

The user may also find useful information and a summary in the following file:

`RayOutput/OutputNumbers.txt`, contains the overall numbers for the assembly

Furthermore, some more information about the de Bruijn graph are located at the following files:

`RayOutput/CoverageDistribution.txt`, The distribution of coverage values

`RayOutput/CoverageDistributionAnalysis.txt`, Analysis of the coverage distribution

`RayOutput/degreeDistribution.txt`, Distribution of ingoing and outgoing degrees

`RayOutput/kmers.txt`, k-mer graph, required option: -write-kmers

The resulting file is not utilised by Ray.

The resulting file is very large.

7. Input Data Profile

At this point are presented the data that are going to be assembled and their characteristics. Before though going further is necessary the understanding of the format of the files that are going to be examined. The common types of input files are the .fasta and .fastq.

The .fasta files have the following format:

```
>gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus]
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPYIGTNLV
```

The first line begins with a greater-than (">") symbol in the first column which is followed by a single line sequence identifier and the description of the library. It is recommended that all lines of text be shorter than 80 characters. When a sequence ends

another one is starting with the exact same routine.

After the first line, follows the sequence in one or more lines with respect to the 80 characters per line as mentioned. The significance of the characters is cited at the appendix due to the size of the information. However the most important are the ACTG for the known common bases. Also if in a given sequence a base is not known, is symbolized with N.

The .fastq files have the following format:

```
@SRR022868.923/1
TTGTTATCCAGTCATTCGTTAGAACTCCTTATAGTACTTATACCNNNNNNNNNNNNNNNNNNNNNNNNNN
+
'I?'F>>$.II.&0@+)I:,6(,2#+*#+#*.'('''+$54%$*!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Again the first single-line is an identifier of the library followed by an optional description. The next line contains the raw sequence data followed by the symbol '+' in another line. The last line contains the encoded form of the quality values for the raw data.

The variety of the characters is a lot simpler and the significance of the characters is the four bases for ACTG. Also with N is symbolized the unknown base.

7.1 Profile of the genomes and their genomic libraries.

7.1.1 Staphylococcus aureus

Genomic size: 2.839.469 bp

Abstract:

Staphylococcus aureus was sequenced using Illumina sequencing technology and then assembled. The libraries were sequenced using short reads and were either fragment or paired end or jumping libraries.

Center Project:

Staphylococcus aureus Assembly Development

Experiment design:	Paired library	Jumping library
Sample:	SRS004752	SRS004751
Run:	SRR022868	SRR022865
Library:	Solexa-8293	Solexa-3932
Avg. Read length:	101bp	37bp
Insert length:	180bp	3500bp
Strategy:	WGS	WGS
Instrument model:	Illumina Genome Analyzer II	Illumina Genome Analyzer II
Run Read Count	1,294,104	3,494,070
Run Base Count	131Mb	129Mb
Coverage	29.1	637.0

7.1.2 **Escherichia coli MG1655** (from now on used as E.coli small)

Genomic size: 4.639.675 bp

Abstract:

Escherichia coli whole genome sequencing with utilization of next generation sequencing technologies.

Center Project:

E.coli Allpaths Assembly Development

Experiment design:	180bp Paired End	**Illumina 5kb Jump Library**
Sample:	SRS009994	SRS269404
Run:	SRR034509	SRR401827
Library:	Solexa-11748	Solexa-44956
Avg. Read length:	101	93
Insert length:	180bp	5000bp
Strategy:	WGS	WGS
Instrument model:	Illumina Genome Analyzer II	Illumina HiSeq 2000
Run Read Count	10,353,618	1,615,703
Run Base Count	2Gb	300Mb
Coverage	212.7	216.0

7.1.3 **Escherichia coli MG1655** (from now on used as E.coli big)

Genomic size: 4.639.675 bp

Abstract:

Escherichia coli whole genome sequencing utilizing next generation sequencing technologies.

Center Project:

E.coli Allpaths Assembly Development

Experiment design:	Fragment Library Construction		Illumina 5kb Jump Library	
Sample:	SRS302375		SRS269404	
Run:	SRR447625	SRR447685	SRR401827	SRR492488
Library:	Solexa-25396		Solexa-44956	Solexa-42866
Avg. Read length:	101		93	93
Insert length:	180		5000	5000
Strategy:	WGS			
Instrument model:	Illumina HiSeq 2000			
Run Read Count	13,479,432	13,457,571	1,615,703	362,200
Run Base Count	2.8G	2.8G	313.4M	67.4M
Coverage	468.3	472.0	110.4	198.4

7.1.4 **Rhodobacter sphaeroides 2.4.1**

Genomic size: 4.607.000 bp

Abstract:

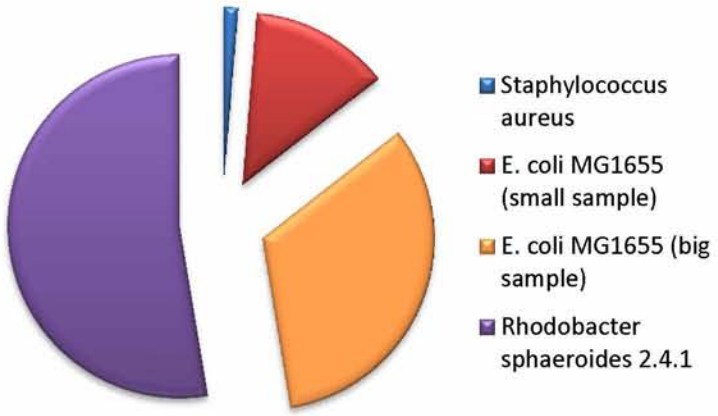
Rhodobacter sphaeroides whole genome sequencing with the use of next generation sequencing technologies.

Center Project:

Rhodobacter sphaeroides Allpaths Assembly Development

Experiment design:	180bp PCR Free Library	4kb Jumping Library	
Sample:	SRS004732	SRS004732	
Run:	SRR125492	SRR034527	SRR034528

Library:	Solexa-11749	Solexa-11767	
Avg. Read length:	101	101	101
Insert length:	180	4000	4000
Strategy:	WGS		
Instrument model:	Illumina Genome Analyzer II		
Run Read Count	11,339,101	17,746,938	20,162,859
Run Base Count	2.3G	3.6G	4.1G
Coverage	153.0	1931.3	2054.9



Left, there is a graphical comparison among the four libraries in order to be clear the difference in their sizes.

7.2 The criteria for the selected libraries

Firstly, also other libraries where tested before the conclusion in these four. The aim was to have a proper amount of libraries of different sizes with various characteristics. To begin with, at a level of genomic size the Staphylococcus aureus is the smallest the two E.coli libraries have the same genomic size and finally, the Rhodobacter has a genomic size on the same level with E.coli.

On a library size level the Staphyloccocus is again the smallest one, with the smallest library coverage. Is a library already tested from the GAGE (http://gage.cbcb.umd.edu/), an evaluation of the very latest large-scale genome assembly algorithms. The specific library was used as a guidance and to test the assemblers with very small libraries.

Next the E.coli MG1655 is a well-known organism with a genomic size double of the Staphylococcus. The E.coli small library has the half of the coverage of the E.coli big library, and it are selected in order to be studied the assemblers on the same organism with two different sizes of data. Moreover, the library size of the E.coli big is

doule the size of E.coli small.

Furthermore, Rhodobacter is the biggest library although the organism by it self has the same genomic size as the E.coli. The theme of this library is that the fragment library has really low coverage and a lot of ambiguities, contrariwise the jumping libraries have really high levels of coverage. The point is to study how the assemblers respond to such difficult input data and if is made right use of the extra information that the jumping libraries supply.

Finally, the size of the Rhodobacter library is bigger than the E.coli big, however the fragment library of E.coli big has the double size of the Rhodobacter library. Moreover, the size of the jumping libraries of the Rhodobacter is greatly bigger than the E.coli big. The aim is to study how the assemblers are going to behave through time and assembly results having a small fragment library and huge jumping information and a big fragmante library with small jumping information.

8. Examination

At this chapter are the results of the assemblers analysis; the assemblers study refers to their efficiency and their performance. The evaluation of the previous characteristics is achieved by the measurement of the size and the depth of the produced contigs and scaffolds, and the resource consumption.

The produced contigs and scaffolds can be evaluated with the following values:

- N50 of contigs (or scaffolds): is the longest length such that at least 50% of all base pairs are contained in contigs of this length or larger. Provides a standard measure of assembly connectivity, higher N50 lengths indicate better performance
- Contigs (or scaffolds) number: the number of the contigs at the end of the assembly should be small. The smaller the number of contigs the better the assembly, always is desired a high value of N50.
- Contigs (or scaffolds)length and distribution. A good criteria is the length of the largest contig and the total length of the contigs

The evaluation of the resource consumption of a certain assembler, consist of examine of the total processing time, and the use of RAM.

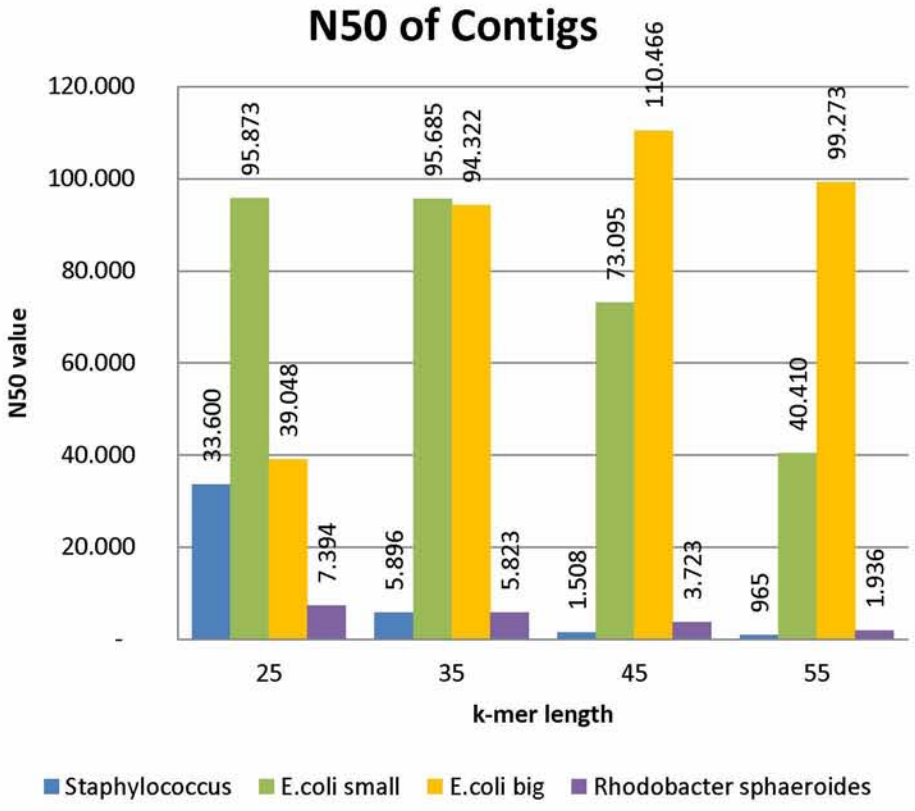
8.1 Ray k-mer evaluation

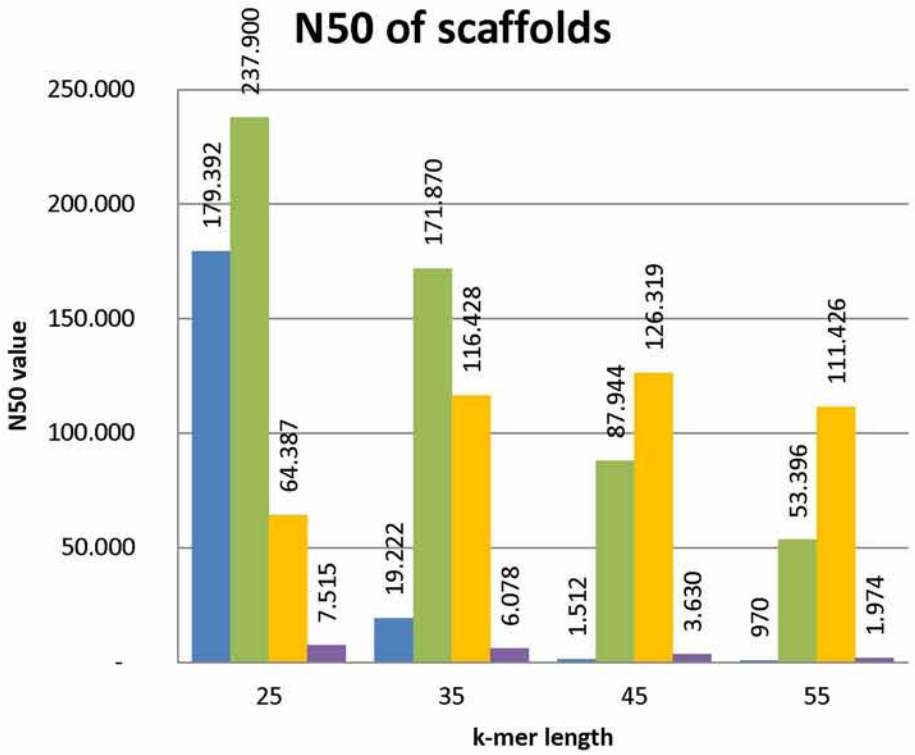
In order to locate the best assembly output of the assembler there where applied two steps, first for each genome, run the assembler for different k-mer values in the range of 25 to 55 with a step of 10. Then a k-mer searching around the area with the highest N50 contigs value, with a step of 2.

After the location of the best k-mer, the assembler was tested running in a range of 16 up to 128 CPUs, such that to study the internal functions time distribution, the

running time, the cost of each running based on the total CPU time and the RAM consumption.

Below is the examination of the N50 values, for the four different samples, of contigs and scaffolds greater than 500bp. For each genome, is analyzed which k-mer length range is going to be examined in parallel with the graph and a sample of the results.





For each genome is presented a sample as a help to the reader, the results with more specific values are at the appendix with the indicated number.

Staphylococcus aureus: The N50 values, along with the population of k-mers, indicate strongly that the most suitable k-mer should have approximate 25bp length. When the length of the k-mer is growing the number of contigs is growing a lot bigger and the same is happening with the number of scaffolds.

k-mer	25	35	*45*	55
Contigs	150	751	1.928	1.962
N50 of contigs	33.600	5.896	1.508	965
Scaffolds	60	559	1.924	1.959
N50 of scaffolds	179.392	19.222	1.512	970

From table st1.

E.coli small: The highest scaffold's N50 is for k-mer length of 25, with the other values of scaffold's N50 enough smaller. So the appropriate area for the selection step could be considered as the one around the 25 k-mer length, but then the fact that the contig's N50 values among the 25 and 35 k-mer length is really close and high, indicate that the best k-mer length sould be between these two sizes. Outside of that range, it is obvious that the longer the k-mer than 35, the smaller the N50. Also between the k-mer size of 25 and 35 the contig and scaffold population stay in the same levels but when the

k-mer size is not in this range the population of them reaches even double values.

k-mer length	**25**	35	45	**55**
Contigs	99	99	132	210
N50 of contigs	95.873	95.685	73.095	40.410
Scaffolds	40	66	108	167
N50 of scaffolds	237.900	171.870	87.944	53.396

From table ecs1.

E.coli big: In this case the results are not so clear, because for the different k-mer lengths the numbers of contig and scaffold N50 and the populations of them do not differentiate significantly enough even though a peak at 45 is caught. In this case, the range of examination in order to get the proper k-mer length was from 35 to 55 with a step of 2.

k-mer length	**25**	**35**	**45**	**55**
Contigs	200	126	96	116
N50 of contigs	39.048	94.322	110.466	99.273
Scaffolds	139	93	88	100
N50 of scaffolds	64.387	116.428	126.319	111.426

From table ecb1.

Rhodobacter sphaeroides 2.4.1: As seen, the results in this case are significally lower than in the other cases and thus not so clear because of the small range. This is due to the large amount of N sequences, thus the library contains a lot of erroneous areas and areas with gaps. The rising values of N50 as the k-mer length is getting smaller and analogous the decreasing number of contigs indicate that the proper k-mer should have length around 25 and maybe should be examined and lower sizes.

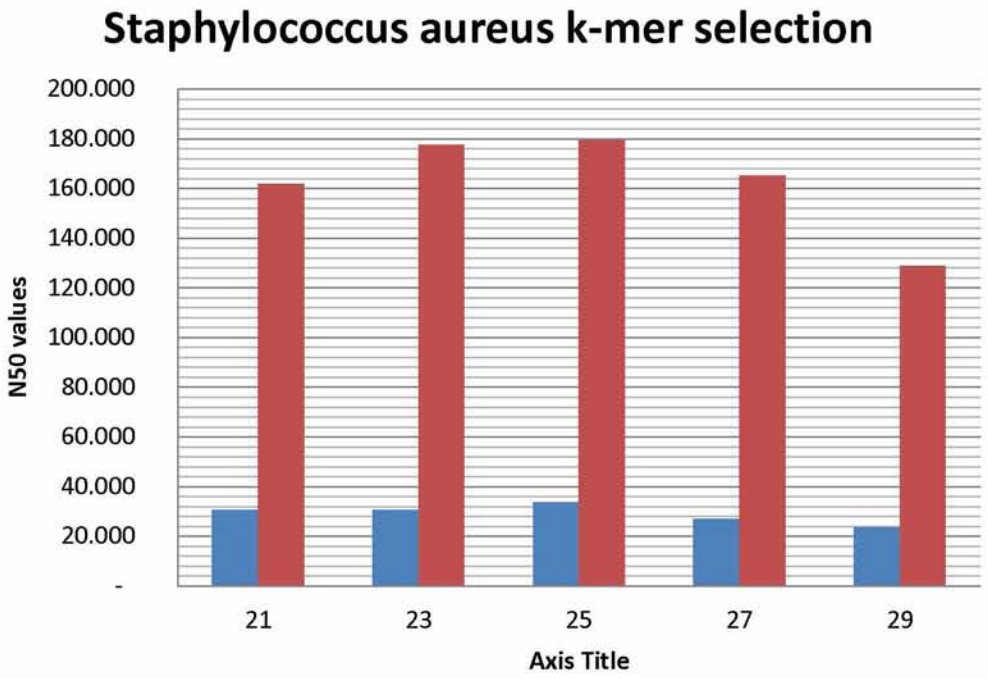
k-mer	25	35	45	55
Contigs >= 500 nt	1.560	1.773	2.111	2.688
N50 of contigs >= 500 nt	7.394	5.823	3.723	1.936
Scaffolds >= 500 nt	1.522	1.736	2.082	2.665
N50	7.515	6.078	3.630	1.974

From table rh1.

Considering all the above, each genome at the next step was tested in the proper range of k-mer length. As seen the k-mer length generally tend to numbers from 25 to 35 but sometimes may go out of that range. So, having the results of the first step, the

next is to focus on a more specific range of k-mer length for each genome library.

At the case of staphylococcus there was a peak at 25 k-mer length. Examining the area around this k-mer length the best results were given with k-mer length of 21, 23 and 25. In the case of 25 kmer length the scaffold and the contigs population is bigger than the 21 and 23, but the point is that the highest N50 of scaffolds and contigs is at 25 k-mer length.

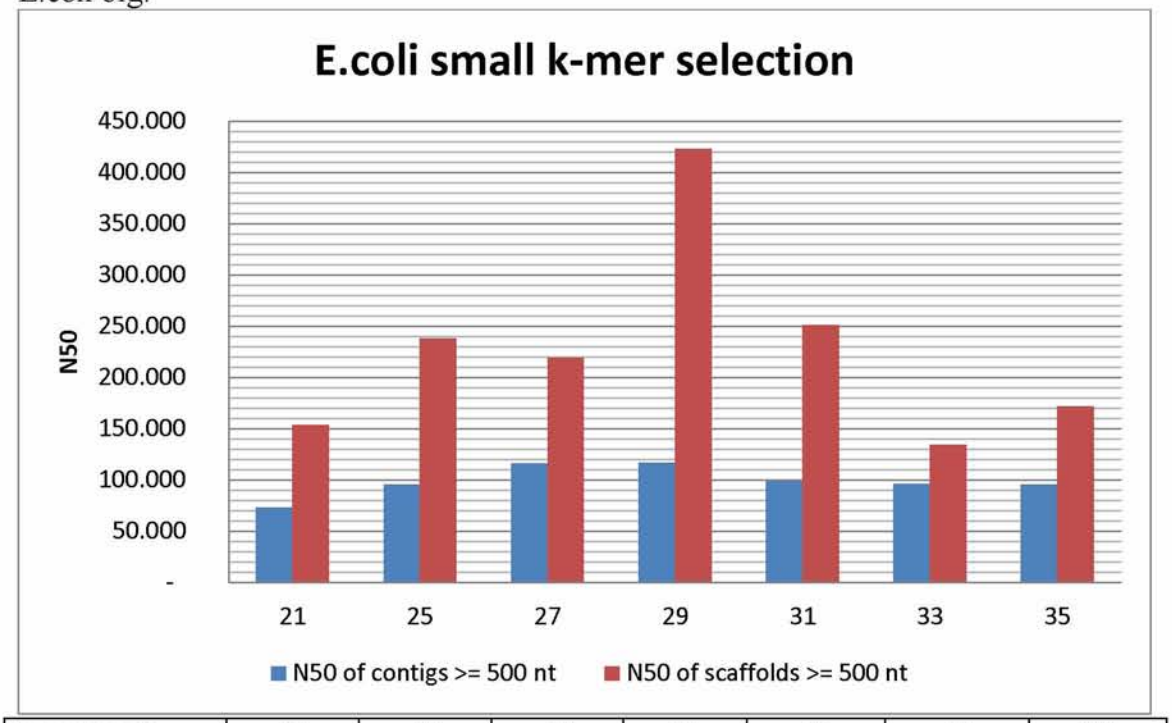


k-mer	21	23	25	27	29
Contigs	149	145	150	173	209
N50 of contigs	30.722	30.730	33.600	26.912	23.570
Scaffolds	52	53	60	61	77
N50 of scaffolds	161.951	177.685	179.392	165.255	128.857

From table st2.

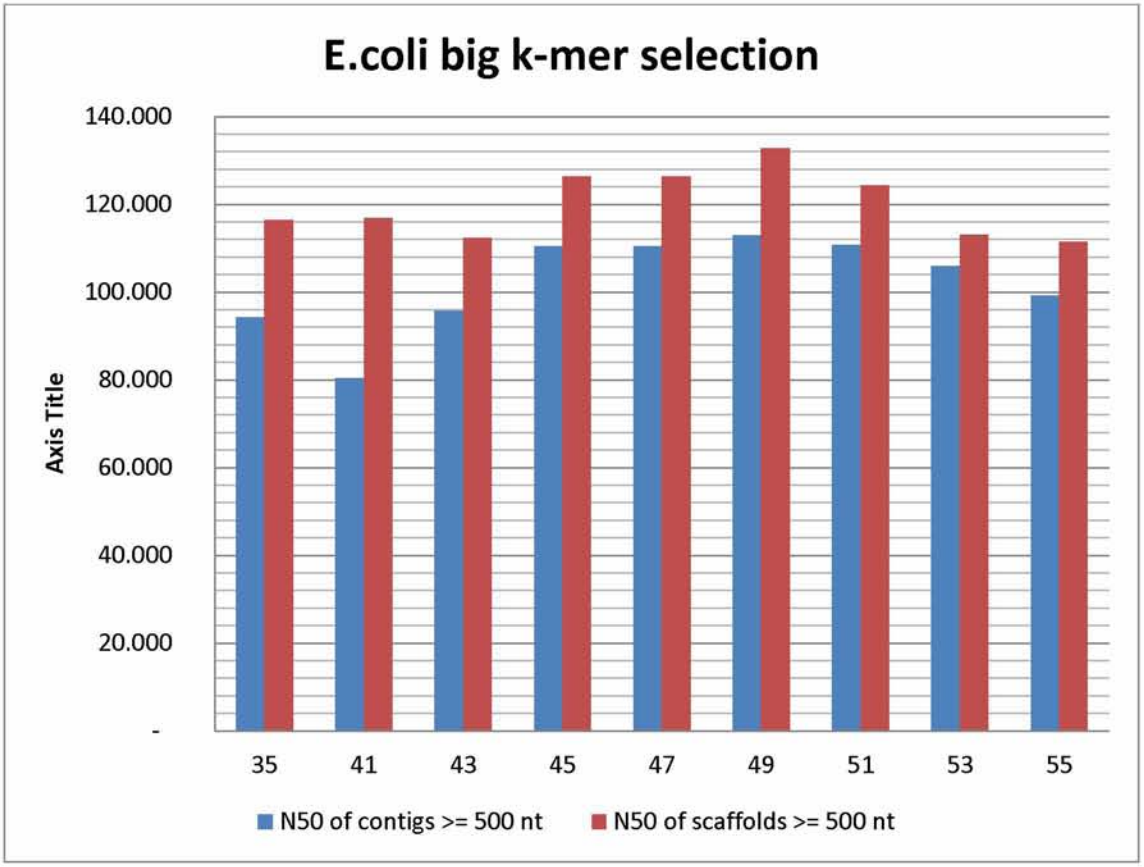
At the case of E.coli small, there was not a peak but an area in which the results where significally higher, so searching in this area the results for kmer length of 29 were by difference the best among the other k-mer lengths. The largest scaffold, as seen for k-mer length in not the largest one, comparing with the value for 31 k-mer length but 29 is preferable because appears N50 with a value double to the others. The largest lengthof scaffolds and contigs is also listed so that a comparison will be done with the

E.coli big.



k-mer length	**21**	**25**	**27**	**29**	***31***	33	**35**
Contigs	118	99	85	83	92	95	99
N50 of contigs	73.198	95.873	116.361	116.738	99.058	96.364	95.685
Largest	239.530	356.997	414.139	414.178	356.801	356.802	356.808
Scaffolds	58	40	50	41	48	59	66
N50 of scaffolds	153.816	237.900	219.424	422.888	251.426	134.554	171.870
Largest	407.484	648.988	535.911	685.259	692.155	664.549	573.799

The E.coli big k-mer selection is located in a lot bigger range as pointed previously; and totally different from the E.coli small, since different fragment libraries are used. Below is a sample of results of the selection step (the results for the k-mer lengths 35, 41,43, 50 are placed in the graph but not in the table for further analysis due to space economy), at the appendix is the whole table. A peak is located for 49 k-mer length with a N50 value of contigns not so higher than the others but with a good enough scaffold's population and N50 comparing with the results for the other k-mer length.

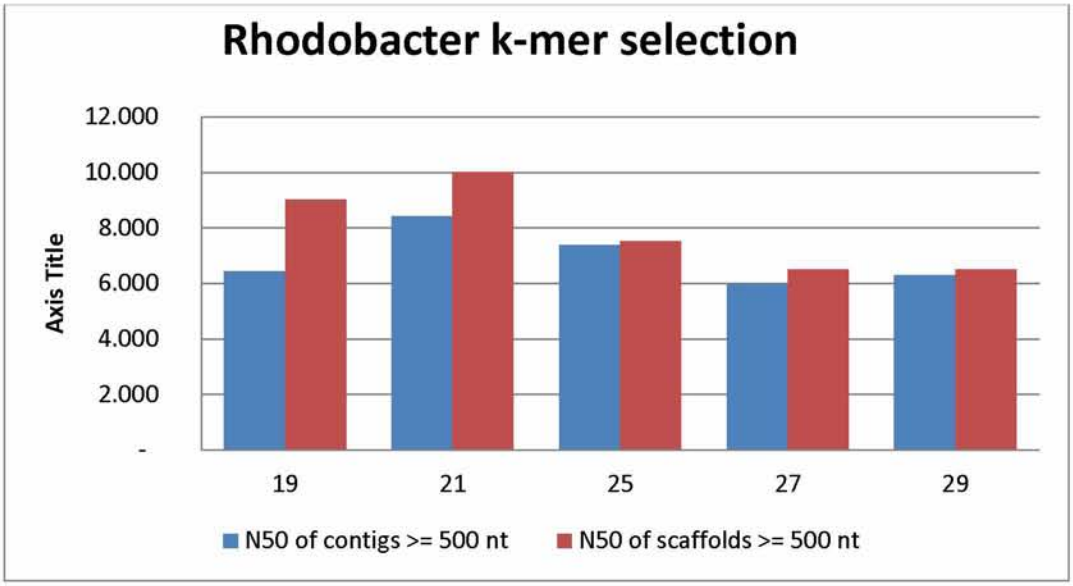


k-mer length	**45**	47	49	51	53
Contigs	96	93	97	99	114
N50 of contigs	110.466	110.466	112.928	110.675	105.950
Largest	414.206	388.518	413.092	388.518	388.518
Scaffolds	88	80	84	82	103
N50 of scaffold	126.319	126.319	132.846	124.342	113.037
Largest	414.206	388.518	413.092	388.518	388.518

The point here is that, comparing the best results from E. coli small with the E.coli big, although the second one has library coverage much bigger, the results from the first one are much more encouraging. At first sight, the N50 of contigs are quite in the same level but this is not happening for the N50 of scaffolds. The same happens with the contig numbers, are fairly same, however the E.coli small population of scaffolds is a lot less and more concentrated. Moreover, comparing the values of the largest contigs it seem that are in the same level, but when it comes to the sizes of the largest scaffolds the assembly of E.coli big do not evolve so much as the E.coli small. Thus, the information of the input should be at the proper amount and not over feed the assembler with information that may confuse him and will prevent him to evolve the assembly. Not always is the larger the better.

Rhodobacter sphaeroides 2.4.1:

Viewing the results of Rhodobacter, are a lot lower than any other case and do not evolve at all. The N50 value of contigs is in the same level with N50 of scaffolds, so the assembly does not develop depth and does not grow. That is because of the high amount of N's in the libraries, and the fact that Ray does not make error correction. Nevertheless, a peak appears for k-mer length 21, but not so clear as seen comparing with the N50 for k-mer length 19. And also the number of scaffolds do not differentiate so much, in order to select the one with the less scaffolds. Also the number of the largest scaffold is bigger with k-mer length 19. That is why a closer look at the results of the total length of scaffolds will help. The total length of scaffolds in case of 19 k-mer length is bigger than the actual genomic size, while with k-mer length 21 the total length is in range.



k-mer	19	21	25	27	29
Contigs	1.617	1.422	1.560	1.602	1.648
N50 of contigs	6.443	8.426	7.394	5.973	6.312
Scaffolds	1.500	1.359	1.522	1.569	1.617
N50 of scaffolds	9.019	10.009	7.515	6.499	6.519
Total length:	4.946.244	4.605.126	4.691.477	4.664.705	4.760.427
Largest	115.145	77.734	92.734	78.911	66.546

Considering, the results from E.coli big and Rhodobacter, it is obvious that Ray does not uses to the fullest the extra information that is given from the jumping libraries. It is noticeable that Ray relies more on the fragment libraries that have a logical coverage.

8.2 ALLPATHLG assembly evaluation

As mentioned before, using ALLPATHs does not require from the user to select a k-mer, instead ALLPATHS declares a general k-mer length 96 and depending the function that the algorithm is running and the stage of assembly changes the k-mer length internally. At this point is appreciated the effort that was done for the libraries preparation, the algorithm does not consume more resources than required for the assembly process.

organism	Staphylococcus aureus	E.coli small	E.coli big	Rhodobacter
contigs>=1000	37	83	93	76
N50 contigs	149.700	95.500	87.800	240.400
scaffolds	10	1	7	54
N50 scaffolds	1.474.400	4.562.000	4.539.000	2.956.000

At the case of E.coli big for ALLPATHSLG are observed the same results as for Ray, the E.coli small have an assembly clearer and more concrete than the E.coli big. Moreover, at the case of Rhodobacter the assembler handles the lack of a good fragment library with the use of the jumping libraries.

8.3 Result Comparison of Ray and ALLPATHSLG

From the results of the ALLPATHLG runs, with ease comes the assumption that have more space and generally the scaffold numbers are very encouraging. The population of the contigs at each organism is significantly smaller in comparison with Ray's. But these results also are affected of the fact that at the case of ALPATHSLG the algorithm gives an output for contigs that their length is greater than 1000 bases; while Ray gives output for contigs that have length greater than 500 bases. Moreover, in every case, ALLPATHSLG presents a more concrete and with bigger depth assembly than Ray. Next, follows a comparison between the results of the two assemblers.

Staphylococcus aureus		
Assembler	**Ray**	**ALLPATHSLG**
Contigs	150	37
N50 Contigs	33.600	149.700
Scaffolds	60	10
N50 Scaffolds	179.392	1.474.400
E.coli small		
Assembler	**Ray**	**ALLPATHSLG**
Contigs	83	83
N50 Contigs	116.738	95.500
Scaffolds	41	1
N50 Scaffolds	422.888	4.562.000

Staphylococcus aureus: At this case ALLPATHSLG has numbers better than Ray, the N50 of scaffolds is the half of the genomic size, while Ray has low numbers.

E.coli small: Ray and ALLPATHSLG have the same number of contigs and Ray presents higher N50 value. But ALLPATHLS evolves the assembly much better and ends up with one unique scaffold that reaches the genomic size of E.coli at a very close point.

E.coli big		
Assembler	**Ray**	**ALLPATHSLG**
Contigs	97	93
N50 Contigs	112.928	87.800
Scaffolds	84	7
N50 Scaffolds	132.846	4.539.000
Rhodobacter		
Assembler	**Ray**	**ALLPATHSLG**
Contigs	1.422	76
N50 Contigs	8.426	240.400
scaffolds	1.359	54
N50 scaffolds	10.009	2.956.000

E.coli big: Again, for Ray and ALLPATHSLG the numbers of contigs range in the same level and Ray presents higher N50 value. However ALLPATHSLG's assembly grows greater with N50 that reaches the genomic size of E.coli.

Rhodobacter: At this case, as mentioned Ray does not manage to handle the big areas of N's and the ambiguities, contrariwise ALLPATHSLG reaches the assembly at a very satisfying level with high numbers

As seen, from the above comparison Ray lacks in the growth of the assembly in the scaffolding level. This is due to the fact that Ray does not perform error correction, losing by this valuable data of the libraries. This is more noticeable at the case of Rhodobacter where the fragment library has many errors and ambiguities. Ray at this case ends up with a number of 25 times up the number of scaffolds of ALLPATHSLG and with a N50 295 times lower.

8.4 Analysis of Time Distribution

At this point, is made an analysis of the time distribution that the algorithms make internally, in relation with the functions that they make use of. It is important when is assigned the assemble of a genome to be known the parts of the assembly that are going to consume more time than the others, the parts that work better in more CPUs and the opposite.

8.4.1 Ray

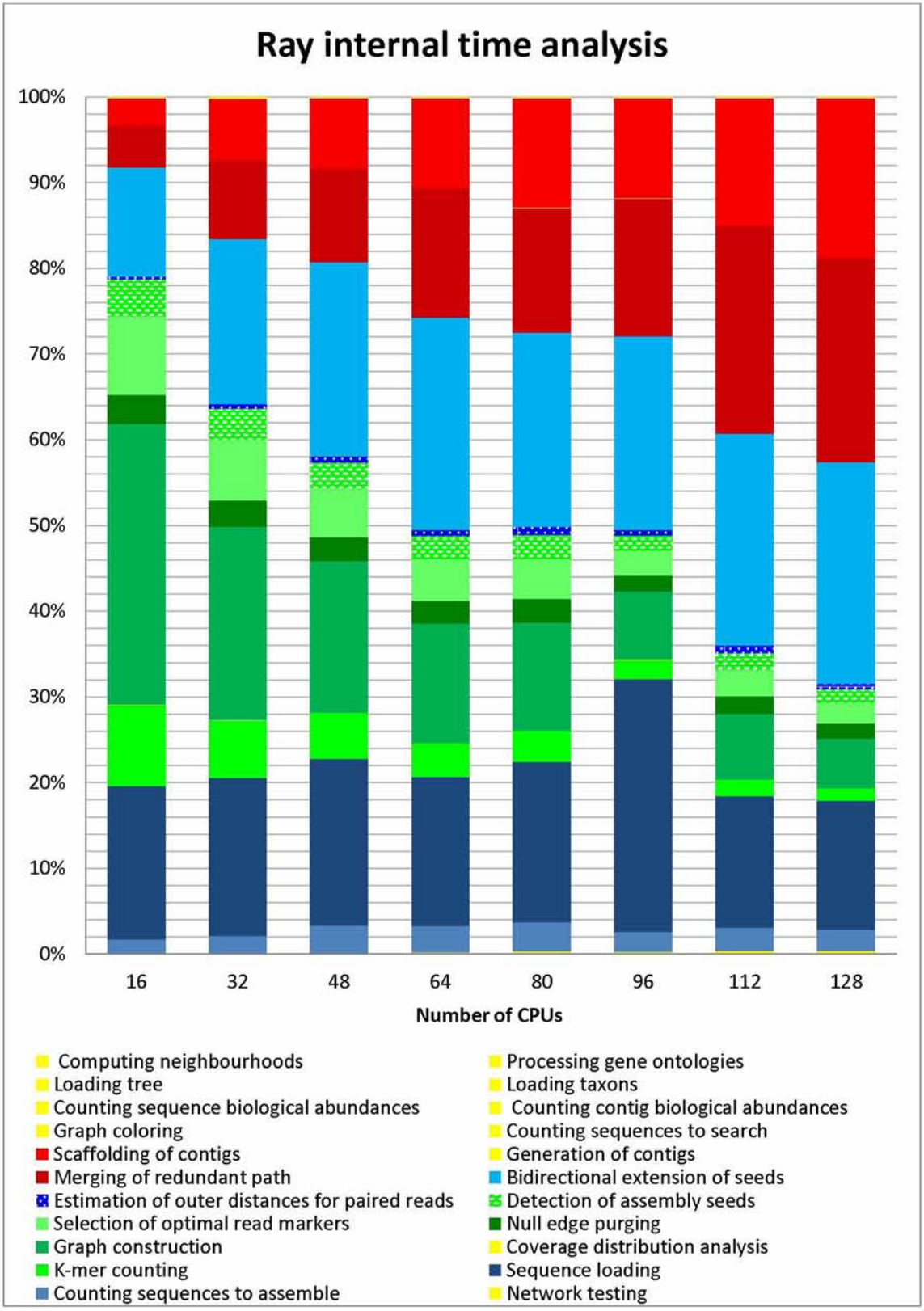
At the graph that follows are contained all the steps- functions that Ray uses during the assembly process. For each number of CPUs used is stated the percentage that a given step is using through the assembly process. By this, is given a general idea how the algorithm distributes the time among his steps and is made clear the pattern of it.At the following graph:

- with yellow shades are mentioned the parts that their time of execution is zero or almost zero and do not affect even slightly the total time. The fact that a lot of them are not present is because the value is zero.

- With blue shades are mentioned the parts that are neutral to the total time and are not affected in a big grade by the number of the CPUs that are used. These steps are scaling more with the size of the provided data.

- With green shades are declared the parts of the algorithm that while the number of CPUs is getting bigger, the smaller is becoming the execution time.

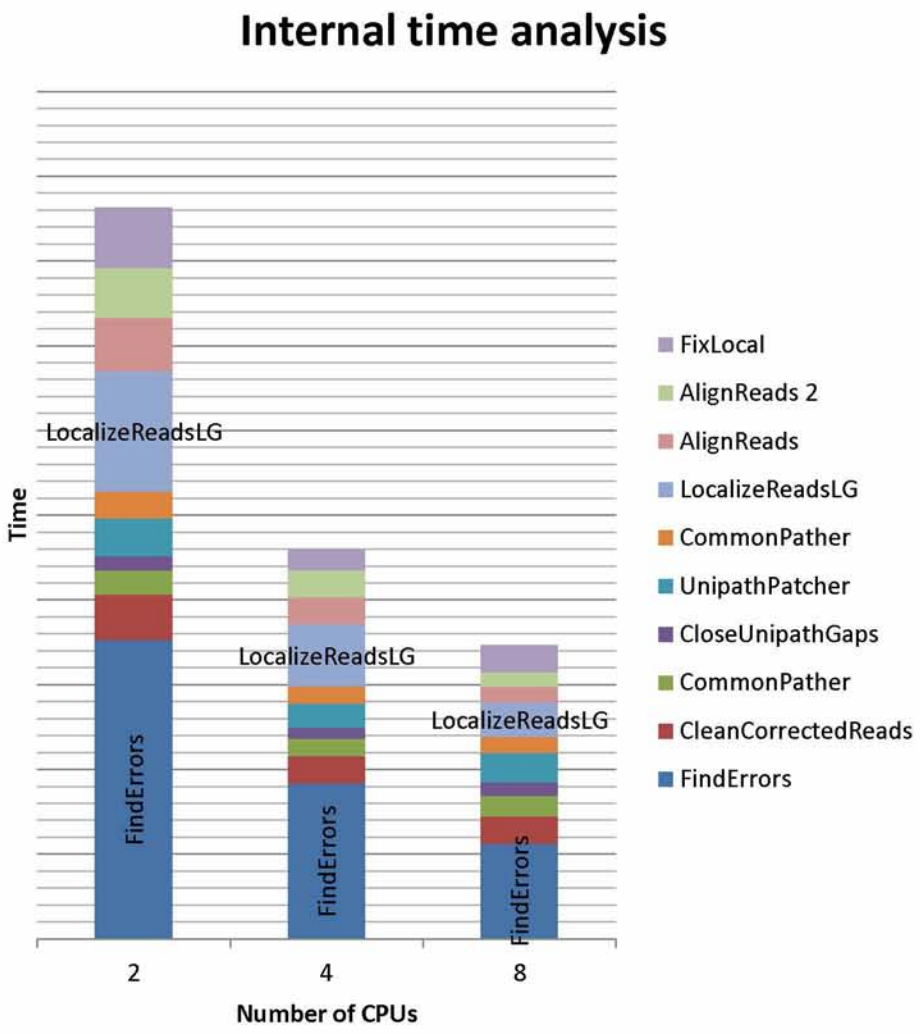
- Finally, with red shades are mentioned the parts that while the number of CPUs is getting bigger the execution time is increased also.


Ray internal time analysis
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
16
32
48
64
80
96
112
128
Number of CPUs
Computing neighbourhoods
Loading tree
Counting sequence biological abundances
Graph coloring
Scaffolding of contigs
Merging of redundant path
Estimation of outer distances for paired reads
Selection of optimal read markers
Graph construction
K-mer counting
Counting sequences to assemble
Processing gene ontologies
Loading taxons
Counting contig biological abundances
Counting sequences to search
Generation of contigs
Bidirectional extension of seeds
Detection of assembly seeds
Null edge purging
Coverage distribution analysis
Sequence loading
Network testing

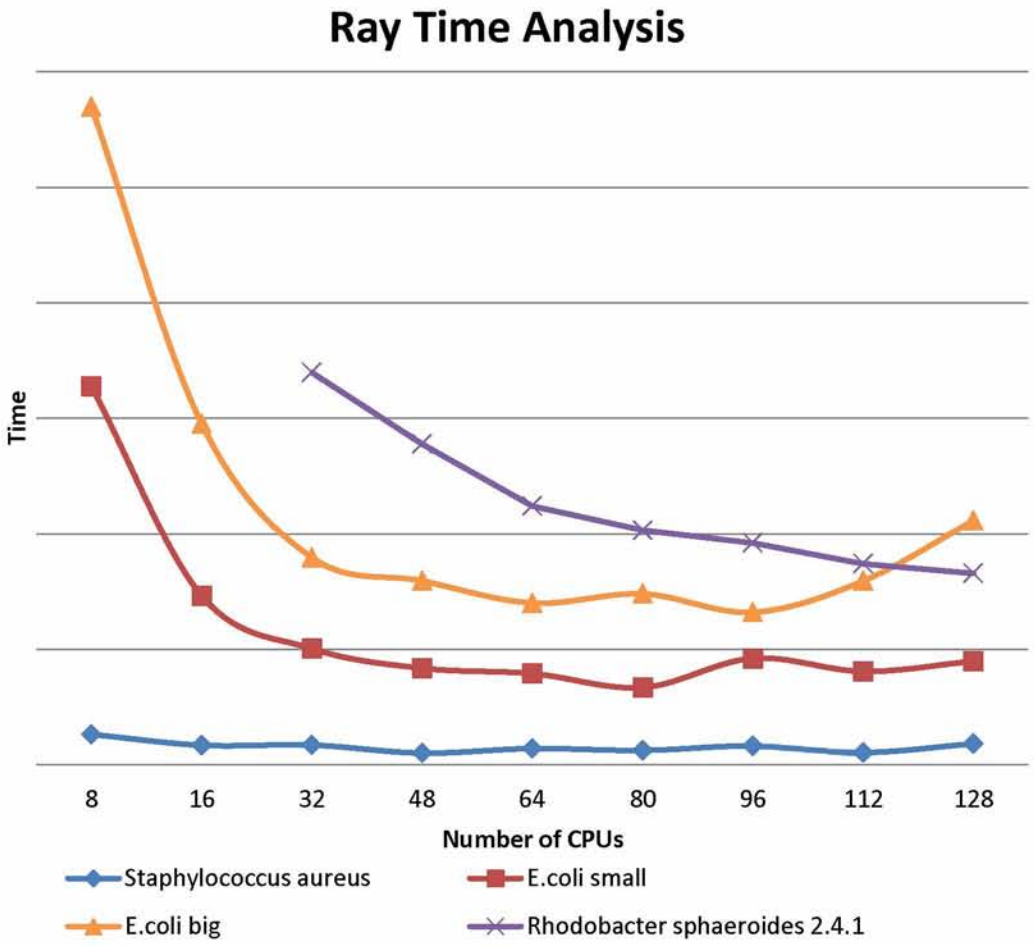
8.4.2 ALLPATHSLG

At the graph that follows are contained only a number of steps- functions that the assembler is using during the assembly process. The steps that are not mentioned is because their execution time is either zero or is not affected in a such grade from the CPU number that is used. Totally ALLPATHSLG passes from 53 stages and here are cited the 10 most time consuming. Next, is presented a pattern that ALLPATHLG follows at the assembly as progresses in relation with the CPUs used. Conversely with Ray, ALLPATHSLG follows a more linear time distribution, and all the steps are reducing while more CPUs are used; which is natural because we are talking about shared memory.

At the most time consuming steps are included the Finderrors, that tries to correct the erroneous reads at the begging of the assembly process and the LocalizeLG that runs an assembly around the selected seed in order to expand the unipaths. Also another observation is that the AlignReads, even after the fixes and the error corrections that are done to the graph with the functions that are between the first and the second execution of AlignReads, the time stays in the same levels.



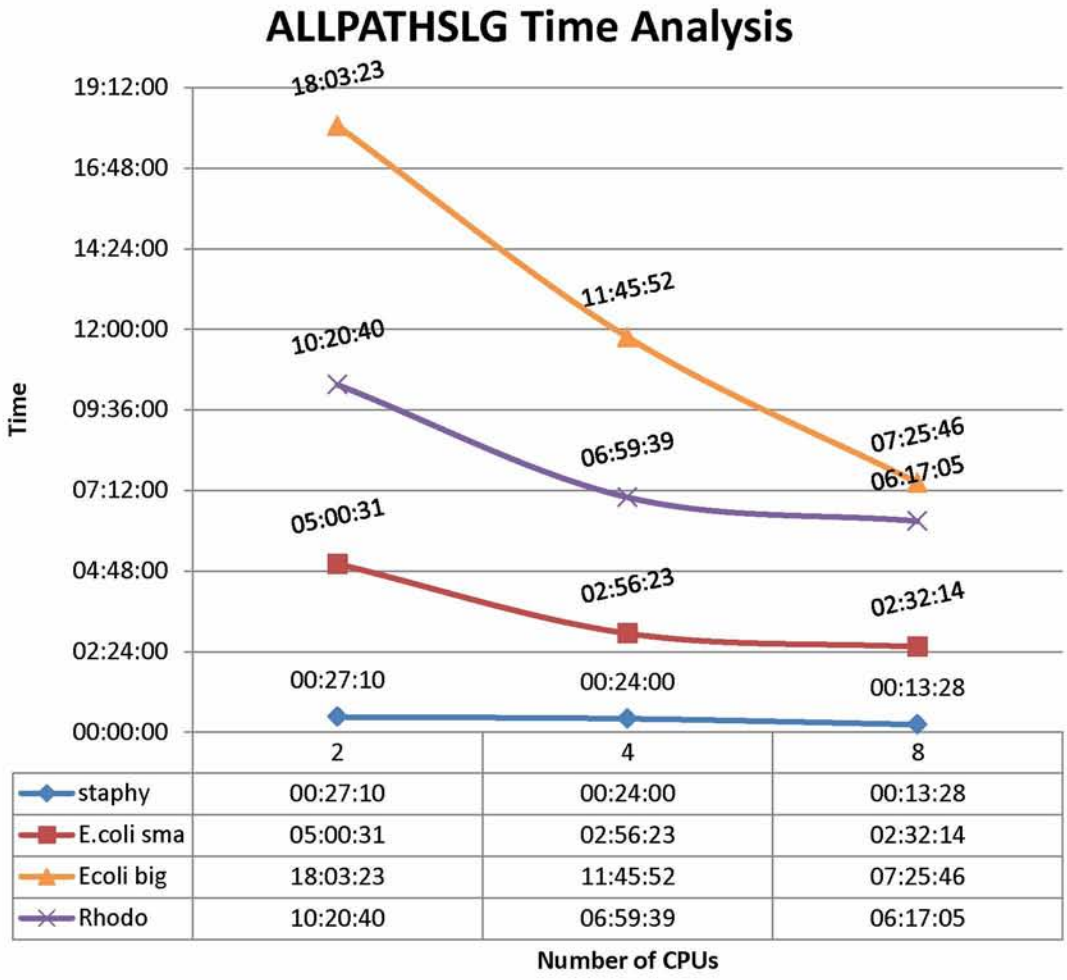
8.5 Analysis of Time in Relation With The Genomic Libraries



Stap/coccus	00:37:49	00:24:20	00:24:31	00:14:33	00:20:14	00:17:53	00:23:26	00:15:16	00:26:11
E.coli small	07:51:49	03:30:39	02:24:44	02:00:17	01:53:37	01:36:16	02:12:38	01:56:19	02:09:01
E.coli big	13:40:19	07:04:50	04:18:16	03:49:07	03:21:34	03:32:57	03:10:01	03:49:01	05:04:14
Rhodobacter	00:00:00	00:00:00	08:08:45	06:39:49	05:22:18	04:52:17	04:36:27	04:10:28	03:58:37

The results of the different runnings that are listed above are indicating that using the proper number of CPUs and not making abuse of the resources give the best results. In the case of Staphylococcus is not so clear, nevertheless using more than 16 cores for a genome of that size is wasteful, even if the best time appears for 3 nodes. Furthermore, for genome libraries with file size from 3Gb up to 20Gb, to get the fastest results a good estimation could be 4 nodes. Although again, 16 nodes would be sufficient because the running time does not differentiate a lot and the cost as will explained later is high. Also, running this size of libraries in more than 4 nodes increases the running time due to the data dependencies and the message passing that is needed. At the case of libraries with total file size bigger than 24-25 Gb Ray should run I more than 1 node, because the memory is not enough to manage the size of the graph. The assembler does not stops but becomes particularly slow because of the lack of

RAM. It will continue but the cost of the assembly will be unusual expensive for this size of genomes. 4 to 5 nodes are enough, the time with more than this number of nodes does not increase significantly.

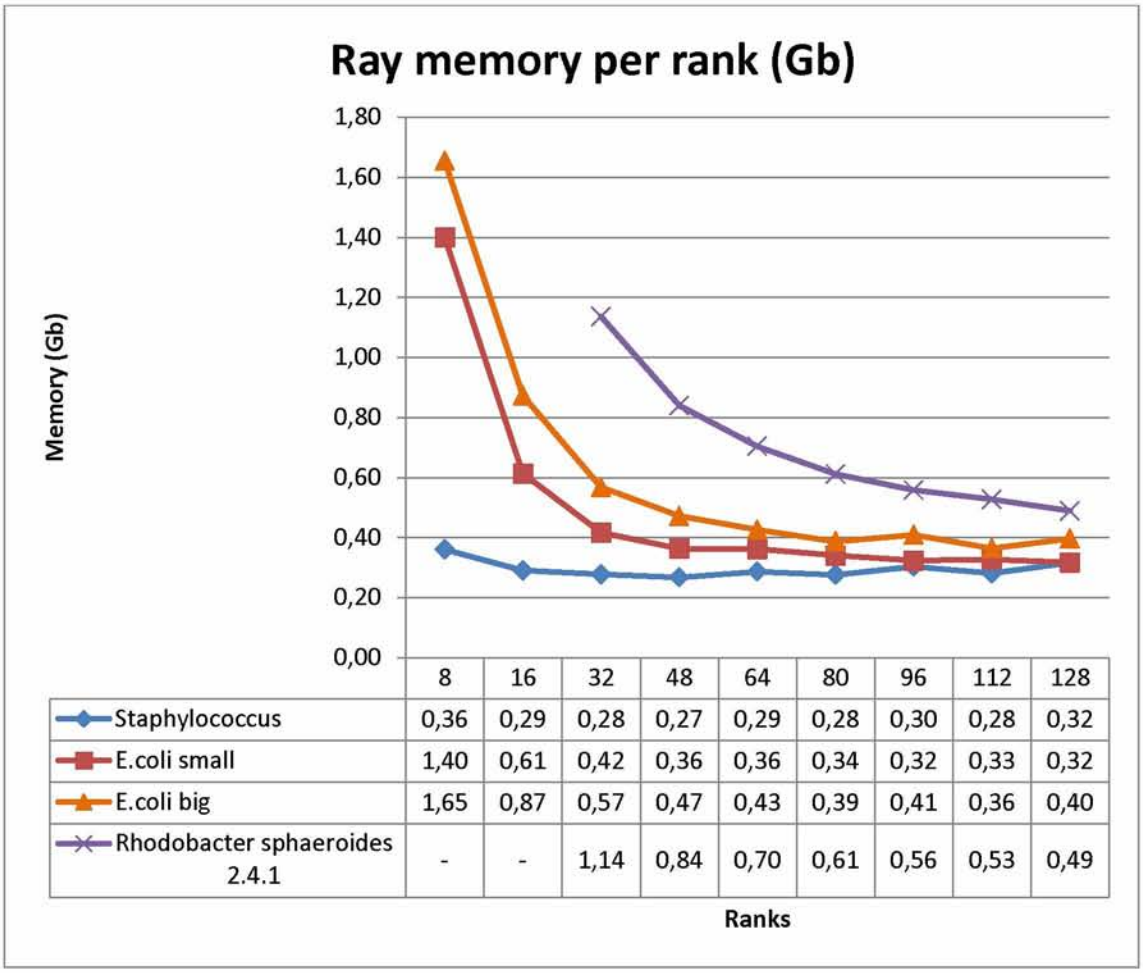


	2	4	8
staphy	00:27:10	00:24:00	00:13:28
E.coli sma	05:00:31	02:56:23	02:32:14
Ecoli big	18:03:23	11:45:52	07:25:46
Rhodo	10:20:40	06:59:39	06:17:05

With shared memory the facts are simpler since all the computations are done in one node. The algorithm reduces the time linearly. Using all the CPUs of the node the time falls almost to the half. At this point, les take a look at the line of Rhodobacter and E.coli big, the expected is the line of Rhodobacter should be above E.coli's, in other words the times of Rhodobacter should be bigger. The fact that is above is due to the detail that E.coli has a fragment library twice the size of the fragment library of the Rhodobacter. So, should be noticed that in order to compute the time consumption should be considered firstly the size of the fragments library.

Furthermore, the algorithm was tested also using one CPU but is not listed due to the fact that the algorithm crushed. A bug was found at the SelectSeeds stage running with 1 thread and reported (the fix is in the next version r42347). Altghough the algorithm could run in a single thread mode would be really slow.

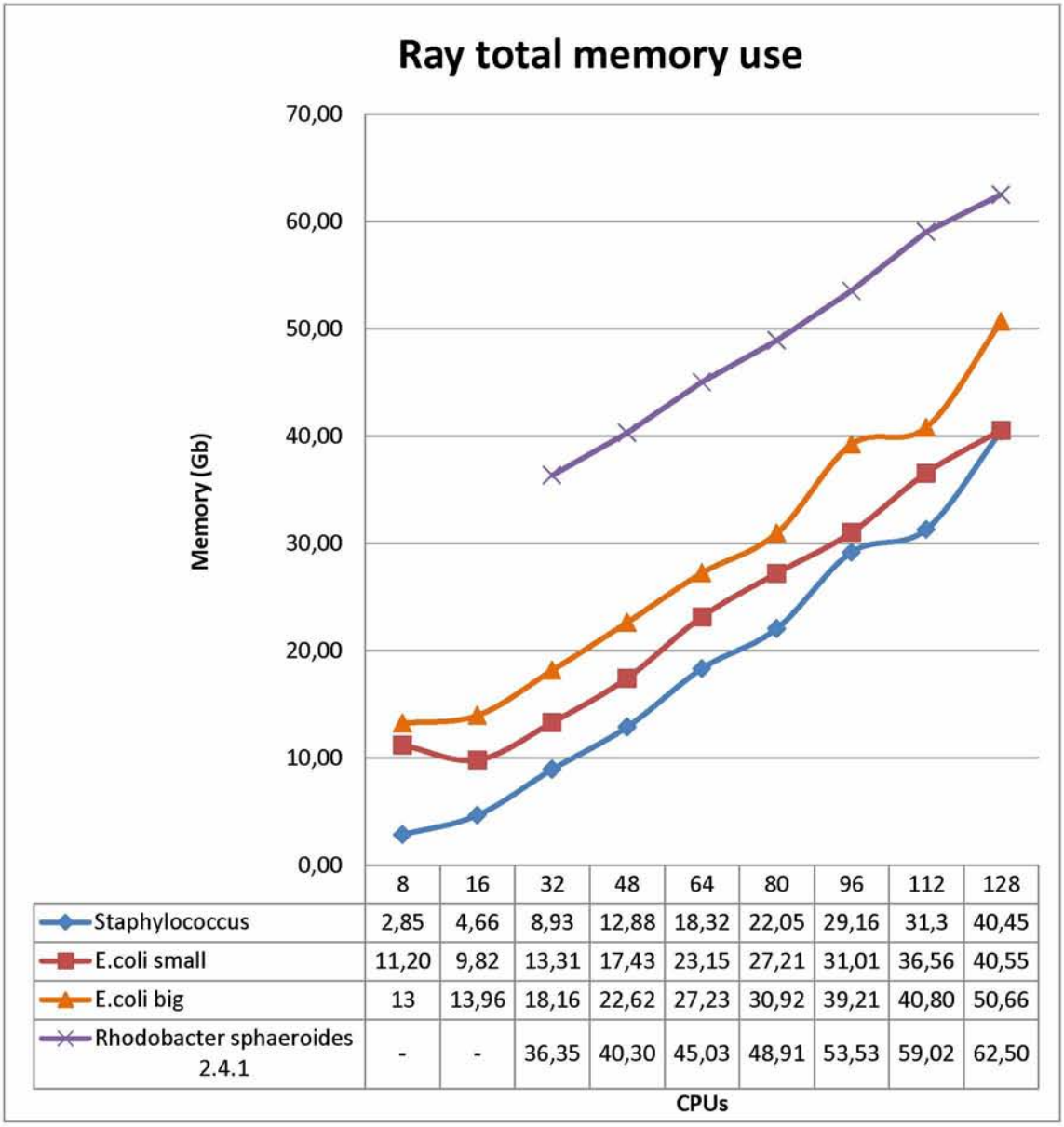
8.6 Memory Analysis



	8	16	32	48	64	80	96	112	128
Staphylococcus	0,36	0,29	0,28	0,27	0,29	0,28	0,30	0,28	0,32
E.coli small	1,40	0,61	0,42	0,36	0,36	0,34	0,32	0,33	0,32
E.coli big	1,65	0,87	0,57	0,47	0,43	0,39	0,41	0,36	0,40
Rhodobacter sphaeroides 2.4.1	-	-	1,14	0,84	0,70	0,61	0,56	0,53	0,49

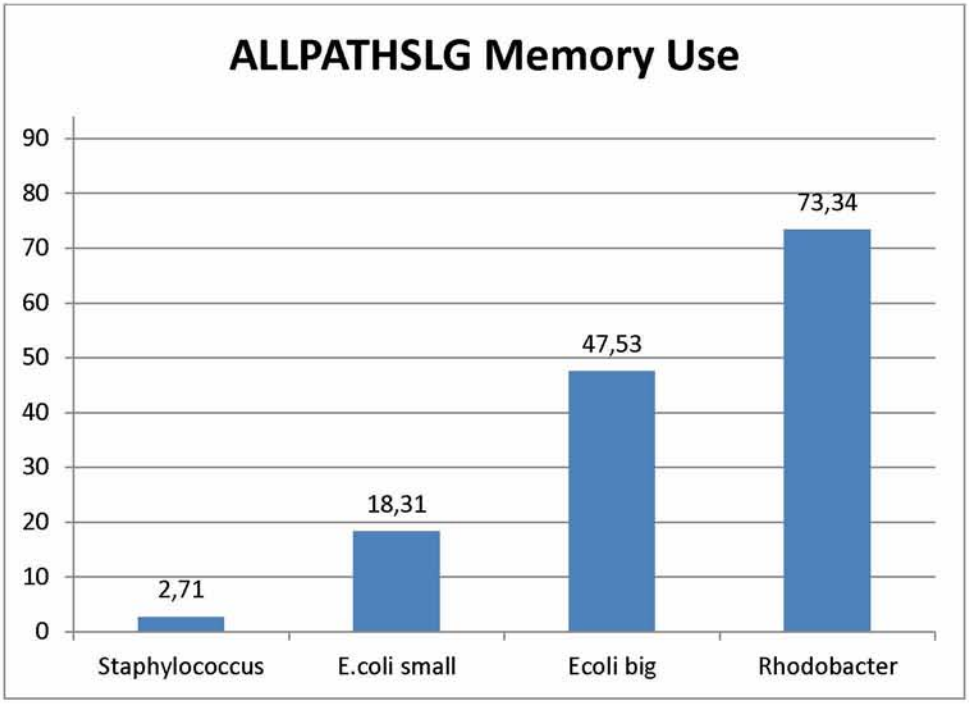
The memory results for the use of Ray were as expected to be, as can be seen form the graphs the memory per rank reduces to the half; except from the case of Staphylococcus because of the significantly small size of the sample. Also due to the size of the Rhodobacter the algorithm could not run with less than 2 ranks.

The point is that for this size of genome libraries the use of 1 to2 nodes is enough to reduce the memory to the half and maintain the speed of the algorithm. Using more than 2 to 3 nodes the use of total memory is raising high levels with no reason and are wasted more resources and the assembly costs more.



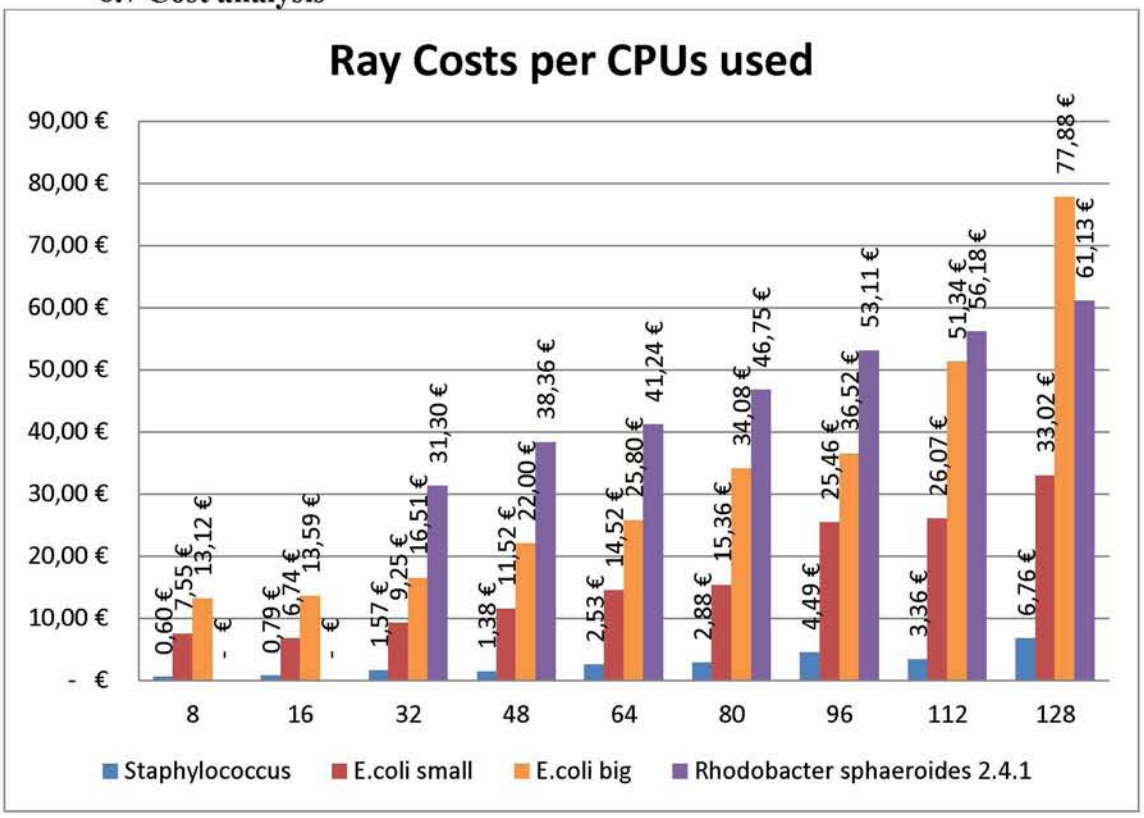
	8	16	32	48	64	80	96	112	128
Staphylococcus	2,85	4,66	8,93	12,88	18,32	22,05	29,16	31,3	40,45
E.coli small	11,20	9,82	13,31	17,43	23,15	27,21	31,01	36,56	40,55
E.coli big	13	13,96	18,16	22,62	27,23	30,92	39,21	40,80	50,66
Rhodobacter sphaeroides 2.4.1	-	-	36,35	40,30	45,03	48,91	53,53	59,02	62,50

CPUs



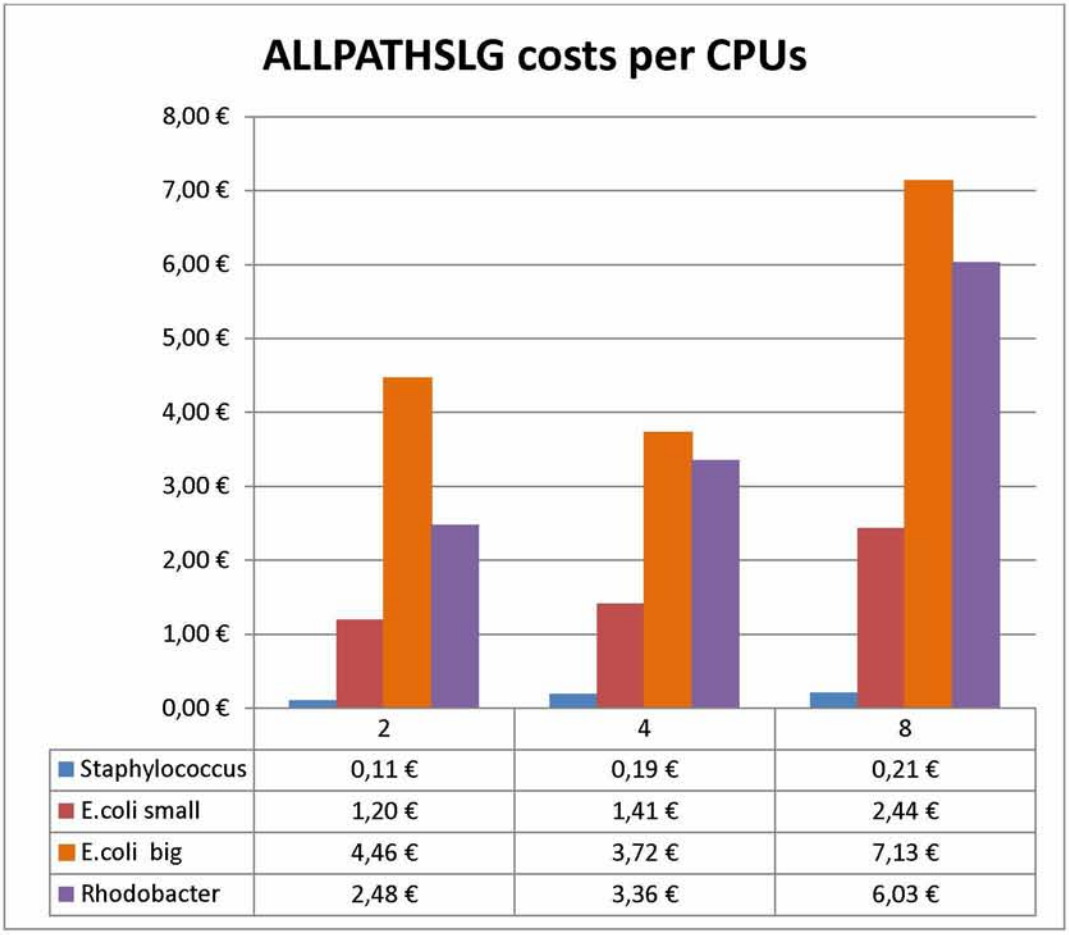
In the case of shared memory, the memory is raising analogous to the size of the genome library. Using ALLPATHSLG the highest RAM usage was almost 80% of the total memory of the node. This gives the prospect that using the intel nodes with ALLPATHLS could be assembled larger genome libraries but not outside of the range of bacteria libraries.

8.7 Cost analysis



Analyzing the cost of use of Ray, the fewer nodes the less cost. The fact is that for this size of libraries heaving results faster costs more. Keeping in mind that every assembly should be a low cost case Ray should be used with the fewer nodes possible. The same happens with ALLPATHS, if the user wants to have a cheap assembly should use the minimum number if CPUs he is allowed. From the results with the use of 2 CPUs the libraries are assembled with the half cost of the assemblies with 8 CPUs .

Comparing ALLPATHS and Ray obviously the first costs less. As the size of the library is getting bigger the difference is significant. For example, using Ray with 2 nodes for Rhodobacter costs 31,30 €, while with ALLPATHS and 2 CPUs of the intel node costs 2,48€. As seen, the use of Ray costs 10 times more.



	2	4	8
Staphylococcus	0,11 €	0,19 €	0,21 €
E.coli small	1,20 €	1,41 €	2,44 €
E.coli big	4,46 €	3,72 €	7,13 €
Rhodobacter	2,48 €	3,36 €	6,03 €

9. Appendix

Ray steps

1	Network testing
2	Counting sequences to assemble
3	Sequence loading
4	K-mer counting
5	Coverage distribution analysis
6	Graph construction
7	Null edge purging
8	Selection of optimal read markers
9	Detection of assembly seeds
10	Estimation of outer distances for paired reads
11	Bidirectional extension of seeds
12	Merging of redundant paths
13	Generation of contigs
14	Scaffolding of contigs
15	Counting sequences to search
16	Counting contig biological abundances
17	Counting sequence biological abundances
18	Loading taxons
19	Loading tree
20	Processing gene ontologies
21	Computing neighbourhoods

ALLPATHSLG steps

1	ValidateAllPathsInputs	Performs basic validation on inputs to ALL-PATHS
2	RemoveDodgyReads	Remove reads that are poly-A.

3	FindErrors	Error correct reads
4	CleanCorrectedReads	Remove reads with low frequency kmers and maybe haploidify
5	PathReads	Path corrected fragment reads
6	FillFragments	Use corrected fragment reads to create filled fragment reads
7	CommonPather	Pathing reads: filled_reads.fastb
8	MakeRcDb	Prepares a data structure (Db) to lookup kmers and kmer paths, generated using the module CommonPather.
9	Unipather	Creating unipaths from filled reads
10	CloseUnipathGaps	Try extending reads to close gaps between unipaths
11	ShaveUnipathGraph	Shave extended unipaths (K40)
12	ReplacePairsStats	Update filtered read stats with new values
13	RemoveDodgyReads	Remove reads that are poly-A or duplicate molecules.
14	SamplePairedReadStats	Use filled fragment unipaths to compute read pair separations
15	UnipathPatcher	Try to patched unipaths
16	CommonPather	Pathing reads: extended.fastb, filled_reads_ext.fastb
17	MakeRcDb	prepares a data structure (Db) to lookup kmers and kmer paths, generated using the module CommonPather.
18	Unipather	Re-creating unipaths for extended unibases
19	FilterPrunedReads	Remove reads containing kmers removed during unipath extending/patching/etc.
20	CreateLookupTab	Create lookup table from unibases
21	ErrorCorrectJump	Correct jumping library reads by aligning to fragment unibases
22	SplitUnibases	Split the unibases into readlike objects
23	MergeReadSets	Merging reads: {filled_reads_filt,unibases_as_reads,jump_reads_ec}
24	MakeRcDb	
25	UnibaseCopyNumber3	Estimating the copy number of unibases (and the corresponding unipaths), based on how many k-mers pile on each unibase.
26	UnipathLocsLG	For building a unipath link graph, aligning reads to unipaths
27	SamplePairedReadDistributions	Use filled fragment unipaths to compute fragment read pair separations
28	BuildUnipathLinkGraphsLG	Building unipath link graph, showing approximate distance between pairs of unipaths

29	SelectSeeds	Select seed unipaths
30	LocalizeReadsLG	Performing an ALLPATHS assembly from the read-estimated unipaths and the error-corrected paired reads
31	MergeNeighborhoods1	Performing an ALLPATHS assembly from the read-estimated unipaths and the error-corrected paired reads, phase 1
32	MergeNeighborhoods2	
33	MergeNeighborhoods3	
34	RecoverUnipaths	recover unipaths that did not make it into the assembly.
35	FlattenHKP	Flatten the assembly HKP into fastavectors, to make a trivial scaffold structure
36	AlignPairsToFasta	Align jumping reads to the assembly fasta.
37	RemoveHighCNAligns	Remove reads that align portion of contigs with CN greater than ploidy
38	MakeScaffoldsLG	Use linking info to build true scaffolds out of a trivial scaffold structure.
39	CleanAssembly	Remove small scaffolds (and/or contigs) from assembly
40	RemodelGaps	Recompute some gaps.
41	PostPatcher	Patch gaps in scaffolds.
42	ApplyGapPatches	Apply gap patches.
43	AlignReads	Build read locations on the assembly for some of the reads.
44	FixPrecompute	Precompute stuff for FixSomeIndels and FixAssemblyBaseErrors
45	FixSomeIndels	Fix minor indel errors in final assembly
46	ApplyAssemblyEdits	Apply edits to the assembly
47	AlignReads	Build read locations on the assembly for some of the reads.
48	FixLocal	Locally reassemble and and attempt to correct errors
49	KPatch	Patch some gaps in the final assembly by walking through the unipath graph
50	TagCircularScaffolds	Identify and tag circular scaffolds
51	RebuildAssemblyFiles	Rebuild final set of assembly files
52	AllPathsReport	Generate basic stats - without a reference
53	LibCoverage	Generate basic library coverage stats - without a reference

10. Bibliography

[1]Next Generation Sequencing. (n.d.). Retrieved 3 6, 2012, 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

[2]Human Genome Project Information. (2009, 10 9). Retrieved 3 4, 2012, from Human Genome Project Information: http://www.ornl.gov/sci/techresources/Human_Genome/project/benefits.shtml

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

[4]Ahmed, M., Ahmad, I., & Khan, S. U. (2010). A comparative analysis of parallel computing approaches for genome assembly. 011 Mar;3(1):57-63. Epub 2011 Mar 3.

[5]Alhakim, A. (n.d.). A Simple Combinatorial Algorithm For De Bruijn Sequences. Potsdam. The American Mathematical Monthly. Vol. 117, No. 8 (October 2010), pp. 728-732

[6]Compeau, P., Pevzner, P., & Tesler, G. (2011). How to apply de Bruijn graphs to genome assembly. Nature Biotechnology 29, 987–991 (2011)

[7]Genomes. DeWeerdt, S. E. (2003, 1 15). Retrieved 3 3, 2012, from Genome News Network: http://www.genomenewsnetwork.org/resources/whats_a_genome/Chp1_1_1.shtml#genome1

[8]Genes. (n.d.). Retrieved 3 4, 2012, from News Medical: http://www.news-medical.net/health/Genes-What-are-Genes.aspx

[9]Genetics Home Reference. (n.d.). Retrieved 3 4, 2012, from Genetics Home Reference: http://ghr.nlm.nih.gov/

[10]JGI Genome Portal. (n.d.). Retrieved 3 5, 2012, from JGI Genome Portal: http://genome.jgi-psf.org/help/index.html

[11]Jones, M. T. (2012, 5 22). Optimizing resource management in supercomputers with SLURM. Retrieved 6 1, 2012, from developersWork: http://www.ibm.com/developerworks/library/l-slurm-utility/

[12]NCBI. (n.d.). Retrieved 3 1, 2012, from A SCience Primer: http://www.ncbi.nlm.nih.gov/About/primer/genetics.html

[13]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)

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

[15]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 Dec;21(12):2224-41. Epub 2011 Sep 16

[16]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).

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

[18]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.

[19] Laboratory of Phil Green, Research. n.d. http://www.phrap.org/index.html (accessed 2012).

[20]Sante Gnerrea, Iain MacCalluma, Dariusz Przybylskia, Filipe J. Ribeiroa, Joshua N. Burtona, Bruce J. Walkera, Ted Sharpea, Giles Halla, Terrance P. Sheaa, Sean Sykesa, Aaron M. Berlina, Daniel Airda, Maura Costelloa, Riza Dazaa, Louise Williamsa, Robert N. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. 2010. PNAS January 25, 2011 vol. 108 no. 4 1513-1518

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

[22]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

[23]Iain MacCallum, Dariusz Przybylski, Sante Gnerre, Joshua Burton, Ilya Shlyakhter, Andreas Gnirke, Joel Malek, Kevin McKernan, Swati Ranade, Terrance P Shea,

Louise Williams, Sarah Young, Chad Nusbaum and David B Jaffe. ALLPATHS 2: small genomes assembled accurately and with highcontinuity from short paired reads. Genome Biology. 10:R103. Epub 2009 Oct 1.

[24]Boisvert, S., Laviolette, F., & Corbeil, J. (2010). Ray: Simultaneous Assembly of Reads from a Mix of High-Throughput Sequencing Technologies. *Journal of Computational Biology*.2010 Nov;17(11):1519-33. Epub 2010 Oct 20.

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

[26]Sovic, I. (s.f.). *Approaches to DNA de novo assembly.* Center for informatics and computing. Zagreb: Ruder Boškovic Institute.

[27] 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

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

[29] Monya Baker. De novo genome assembly: what every biologist should know. Nature Methods 9, 333–337 (2012) doi:10.1038/nmeth.1935