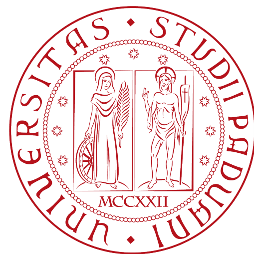
A maximum likelihood approach to genome assembly

Laureando: Giacomo Baruzzo
Relatore: Prof. Gianfranco Bilardi

08/10/2013



UNIVERSITÀ DEGLI STUDI DI PADOVA
Dipartimento di Ingegneria dell'Informazione - DEI
Corso di Laurea Magistrale in Ingegneria Informatica

A.A. 2012-2013

“Il DNA dell’uomo differisce per solo il
2% da quello degli scimpanzè.
Ma in quel misero 2% ci sono tutti
gli Einstein e i Leonardo,
gli Chopin e i Van Gogh”

Abstract

DNA research is one of the most important fields of study in biology and medicine. The first step in DNA research is sequencing, *i.e.* the process of determining the precise order of nucleotides within a DNA molecule. There are several DNA sequencing processes, the last one in chronological order is called NGS (Next Generation Sequencing). This method arose from the high demand of low-cost and high-throughput sequencing and its usage spread in the latest years. Unfortunately, current sequencing technology cannot read the entire genome in one single iteration, so small fragments (called reads) are produced and stored for future assembly. DNA assembly is the bioinformatics' problem to reconstruct the original molecule from its sub-sequences, produced by a sequencing process. *De novo* assembly refers to a particular case where the reconstruction of original sequence is done with no previous knowledge on sequence. The assembly algorithms are typically complex and with a high computational complexity. For this reason commercial softwares use heuristics and custom data structures in order to reduce such complexity. Currently, graph algorithms are the most popular ones. Recently, a new framework based on maximum likelihood approach has been proposed for post-processing graph algorithms data. Inspired by maximum likelihood approach, a new experimental probabilistic approach to de novo whole genome assembly was developed. The current formulation for the model works for NGS data, but it can be adapted for data of previous generations either.

In this thesis, for the first time this new stochastic approach has been implemented into a software assembler. Starting from the given model, a parallel software has been developed in order to obtain a first experimental validation of the model. Finally, some artificial data have been tested and the results of simulations have been analyzed to study correctness and reliability of the experimental model.

Contents

Abstract **5**

1. Introduction **7**

1.1. Thesis Outline . . . 8

2. DNA sequencing and genome assembly **9**

2.1. Sequencing . . . 9

2.1.1. Phred quality score . . . 10

2.1.2. The NGS method . . . 11

2.1.3. Coverage . . . 11

2.2. De novo assembly . . . 12

2.2.1. Greedy . . . 13

2.2.2. Overlap Layout Consensus (OLC) . . . 13

2.2.3. De Bruijn Graph (DBG) . . . 13

2.2.4. Maximum likelihood genome assembly . . . 14

2.2.5. Pro and cons of current approach . . . 16

3. Our probabilistic approach **17**

3.1. Introduction to the stochastic model . . . 17

3.2. The basic idea . . . 18

3.2.1. Read set posterior probability and maximum likelihood principle 18

3.2.2. Single pair probability . . . 19

3.2.3. Single read probability . . . 19

3.2.4. Positioned read posterior probability given the sequence . . . 21

3.2.5. Single read posterior probability given the sequence . . . 22

3.2.6. The final formulation . . . 22

3.2.7. Read set probability . . . 23

3.3. Comparison with current approaches . . . 24

4. The program development **25**

4.1. Program goals . . . 25

4.2. The stochastic model into the software . . . 26

4.3. Challenges, implementation choices and platform . . . 26

4.3.1. Exponential problem . . . 26

4.3.1.1. The C Programming Language . . . 27

4.3.1.2. POSIX Pthread . . . 27

4.3.1.3. IBM P770 Power7 28
4.3.1.4. AIX, LoadLeveler and XL C compiler 28
4.3.2. Numeric precision 29
4.3.2.1. Custom floating point data type 29
4.3.2.2. Formulas simplification 32
4.4. The assembler - Description and pseudocode 33
4.4.1. Input and parameters setting 34
4.4.2. Probability computation 34
4.4.3. Ranking 34
4.4.4. Output 36
4.5. Code optimization solutions 37
4.5.1. General best practices adopted 37
4.5.2. Specific code optimization 38
4.6. Compiler optimization and tuning 39

5. Simulations and experiments **41**
5.1. Modus operandi 41
5.2. Ranking 42
5.2.1. Correctness 42
5.2.2. Reliability 50
5.2.3. Robustness 51
5.3. Read set 53

6. Conclusion and future work **55**

A. Source code organization **59**
A.1. Data type 59
A.2. Utilities 59
A.3. Probability function 60
A.4. Assembler 60

B. POSIX Pthread functions **61**
B.1. Thread creation 61
B.2. Thread attribute 62
B.3. Thread termination 62
B.4. Thread join 63

Bibliography **65**

Acknowledgments **67**

List of Algorithms

4.1. Assembler 33
4.2. Thread probability computation 34
4.3. Thread merge operation 36

List of Figures

2.1. DNA sequencing and assembly 9

4.1. Merge multithread with NT=4 threads 35
4.2. Compiler optimization 39

5.1. True sequence ranking positions 50

List of Tables

5.1. Test Ranking (TOP 20): C≃3.14; N=14; M=11; m=4; 0 base errors . 42
5.2. Test Ranking (TOP 10): C=3.2; N=15; M=12; m=4; 1 base errors . 43
5.3. Test Ranking (TOP 10): C=5.3; N=15; M=20; m=4; 0 base errors . 44
5.4. Test Ranking (TOP 10): C=5.3; N=15; M=20; m=4; 3 base errors . 45
5.5. Test Ranking (TOP 10): C=5.23; N=13; M=17; m=4; 2 base errors 46
5.6. Test Ranking (TOP 10): C=5; N=12; M=15; m=4; 1 base errors . . 47
5.7. Test Ranking (TOP 10): C≃10.66; N=15; M=40; m=4; 1 base errors 48
5.8. Test Ranking (TOP 10): C≃10; N=14; M=35; m=4; 0 base errors . 48
5.9. Test Ranking: C≃5; N=15; M=20; m=4 51
5.10. Test Ranking: C=20; N=13; M=65; m=4 52
5.11. Test Read Set: C≃5; N=15; M=20; m=4 53
5.12. Test Read Set: C=20; N=13; M=65; m=4 54

1. Introduction

In the last 30 years, DNA research has been one of the most important fields of study in both biology and medicine. The genome contains all the biological information of an organism and all this data are encoded in DNA or RNA sequences. The entire genome length varies from a few thousand of base pairs in simple viruses to hundreds of billions in more complex organisms.

From the first experiment in the 1970s to current studies, available technologies and biology knowledge have evolved significantly. Mathematics, computer science and engineering provide powerful tools for biological data analysis and new interdisciplinary fields have arisen, like bioinformatics.

The first phase in DNA studies is sequencing, the process of determining the precise order of nucleotides within a DNA molecule. Current sequencing technologies cannot read the entire genome in one single iteration, so small fragments (called reads) are produced and stored for future assembly. Research in this area tries to reduce sequencing cost, elaboration time while increasing reliability. There are several DNA sequencing processes, the last one in chronological order being called NGS (Next Generation Sequencing). This method was born from the high demand of low-cost and high-throughput sequencing and has been raising in the latest years. NGS data typically present larger amount of reads, shorter read lengths, higher coverage, and different error profiles compared with the previous generation sequencing data. The main advantage of NGS method and the reason for its diffusion is a drastic reduction in sequencing cost.

The output of sequencing process is elaborated by some pieces of software, called assemblers. DNA assembly is the bioinformatics' problem of reconstructing the original molecule from sub-sequences, produced by a sequencing process. De novo assembly refers to a particular case where the reconstruction of the original sequence is done with no previous knowledge on the sequence itself. Different methods have been proposed in the latest years to solve this problem and many software assemblers have been developed. Unfortunately, assembly algorithms are typically complex and with high computational complexity. Also, large volumes of data and high number of operation require often high-performance computing platforms. In order to mitigate these problems, commercial softwares employ heuristics and custom data structures to reduce the complexity. In fact, currently most popular algorithms use graphs to resolve assembly problem. However, recently a new framework that utilizes a maximum likelihood approach has been proposed for post-processing of graph algorithms data.

Inspired by maximum likelihood approach, a new experimental probabilistic approach to de novo whole genome assembly has been developed. Current model formulation works with NGS data, but can be adapted for previous generation data. The main difference with current approaches is that this new model propose an exact and exhaustive approach, using a solid stochastic formulation of assembly problem instead of current graph's heuristics. Another important features is that all produced results are measurable, since for all sequences tested a probability/likelihood values is produced.

1.1. Thesis Outline

In this thesis, for the first time this new stochastic approach has been implemented into a software assembler. Starting from the given model, a parallel software has been developed in order to obtain the first experimental validation of the model. Then some artificial data have been tested and the results of simulations have been analyzed to assess correctness and reliability of the model.

The dissertation starts in *Chapter 2*, where is presented a brief description of the sequencing process, a formulation of the assembly problem and an overview of the current approaches to the problem.

In *Chapter 3* the used stochastic model is exposed, explaining the maximum likelihood problem formulation and listing used assumptions and hypotheses.

Preliminary analysis, implementation choices and main challenges in software development are presented in *Chapter 4*. Also the implementation and software features are described.

In *Chapter 5*, the performed experiments are reported, with some considerations and observations on correctness and reliability of the result data.

Conclusions and points of interest for future developments are summarized in *Chapter 6*.

2. DNA sequencing and genome assembly

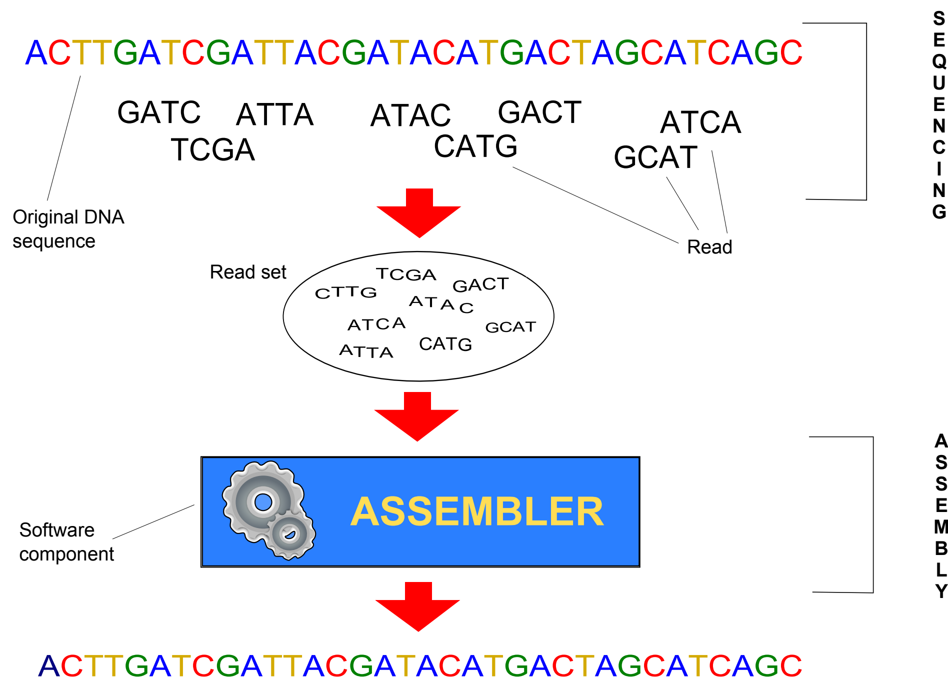


Figure 2.1.: DNA sequencing and assembly

2.1. Sequencing

DNA sequencing is the process of determining the precise order of nucleotides within a DNA molecule. Most DNA molecules are double-stranded helices, long sequences of the four bases nucleotides (adenine (A), cytosine (C), guanine (G) and thymine (T)). The genome contains all the biological information of an organism and all this data are encoded in a DNA or RNA sequence. The entire genome length varies from a few thousand of base pairs in simple viruses to hundreds of billions in more complex organisms.

Though DNA studies began in 1950s, only in 1970s the first fragments of DNA could be sequenced in laboratory. During the 1980' and 1990' several methods and

machines were developed for both academic and commercial purposes. In 2000 the Human Genome Project identified and mapped the human genome for the first time. Nowadays, DNA research represents one of the most important areas of study in medicine and biology. The cost reduction provided by the advent of fast DNA sequencing methods allowed research to accelerate and grow.

Current sequencing technologies cannot read the entire genome in a single iteration, so small fragments (called reads) are read and stored for future assembly. Reads are sequences of single-letter base, subparts of original DNA sequence, whose length values are in the range 20 – 1000 bases. The different technologies often differs for reads length. Research in this area tries to reduce sequencing cost, elaboration time and increase results reliability.

2.1.1. Phred quality score

In the sequencing process, often some numeric values are associated to elaborated bases. These values tell some information about base reliability. A widely accepted quality value score is Phred, originally developed by Phred program in the Human Genome Project. Phred quality scores are assigned to each nucleotide base, in the same automatic process that generates the reads. Main functions of Phred quality value score are:

- In assembly preprocessing, phred quality value score can be used to remove/-manage low-quality reads.
- In assembly elaboration, phred quality value score can be used to choose between multiple assembly options.
- In assembly post-processing, phred quality value score can be used to assess the quality of produced sequence.

In the phred schema quality values set is given by $Q = [0, +\infty]$ and the error probability function is defined as

$$P_e(q) = 10^{-\frac{q}{10}} \quad \forall q \in Q$$

For example, a Phred quality score $q = 10$ for a base means that base accuracy is 90%.

In all practical applications, the set $Q = [0, Q_{max}]$ is indeed finite. To guarantee that P_e remains a properly defined probability function it is possible to define the error probability as:

$$P_e(Q_{max}) = \sum_{q=Q_{max}}^{+\infty} 10^{\frac{q}{10}}$$

2.1.2. The NGS method

There are several DNA sequencing processes, the last one in chronological order being called NGS (Next Generation Sequencing). This method was born from the high demand of low-cost and high-throughput sequencing and has been raising in the latest years.

NGS data typically present a larger amount of reads, shorter read lengths, higher coverage (sec. 2.1.3), and different error profiles compared with previous generation sequencing data. Short read (today range is about 25-400 bp) deliver less information and requires higher coverage allowing a better assembly but increasing complexity and computational effort. These drawbacks are balanced by the dramatic reduction of sequencing costs which is one of the main advantage of NGS method as well as one of the reason for its success.

In data production some reads were produced by breaking up randomly the original DNA sequence in various small segments. This process is called shotgun process. Associate with each single base in the read often there is a numeric quality value which offers information about reliability of single base. Today, only some NGS assembly softwares exploit these parameters.

2.1.3. Coverage

In sequencing, an important parameter is *coverage*. Coverage is an index than combine genome length N, number of read M and read length m.

$$C = \frac{M \cdot m}{N}$$

The coverage is the average number of reads that represent a base in the original sequence and gives a measure of redundancy in data production. Note that coverage is only a statistical index: each base can have a different real coverage. A high coverage in *shotgun sequencing* (sec. 2.2) is desired because it can overcome errors in base calling and assembly. NGS data have greater coverage than old generation sequencing data.

2.2. De novo assembly

DNA assembly is the bioinformatics' problem to reconstruct an original molecule from its subsequences (reads), produced by a sequencing process. De novo assembly refers to a particular case where the reconstruction of the original sequence is done with no previous knowledge on the sequence. The larger amount of data, as well as the shorter reads introduced in the latest years have given new life to research in shotgun assembly algorithms.

The assembly process is possible when the target sequence is over-sampled, such that reads overlap in some part. The information given by the overlap lead algorithm decision and allow the assembly. However, all DNA sequencing technologies have the fundamental limitation that read length is much shorter than the target sequence. NGS data, for example, try to reduce this problem with large amount of reads and high coverage. One of the main challenges in assembly phase is to manage repeat sequence in the target. DNA regions that share repetitions can be indistinguishable, mostly if repetitions are longer than reads. Unfortunately, the short read used in NGS concurs to this problem.

The errors present in data must be managed by assembler softwares, allowing imperfect alignment to avoid missing true match. On the other hand, an excessive error tolerance leads to false positive match. All software products have peculiar error balance control and generally some preprocessing of data is done. Also, assembly accuracy is difficult to measure: some indices (the N50 statistic for example) exist but require some constraints to compare different assemblies.

Unfortunately, assembly algorithms are typically complex and with a high computational complexity. Also, large volumes of data and an elevated number of operations require often high-performance computing platforms. In order to mitigate these problems, commercial softwares use heuristics and custom data structures to reduce the complexity. As a consequence, currently most popular algorithms use graphs to resolve assembly problem. Three main categories of graph algorithm exist: Overlap/Layout/Consensus (OLC) methods, de Bruijn Graph (DBG) methods and greedy graph methods.

In currents graph algorithms, assembly is a graph reduction problem and most of these reduction belong to NP-hard class. In [9] Nagarajan and Pop showed that assembly problem on overlap graph and on de Bruijn Graph is a NP-hard problem. Only with very high coverage and repetitions in the original genome with some length property, a polynomial formulation can be reached.

2.2.1. Greedy

Greedy algorithms were the first algorithms developed for NGS assembly. The base operation in a greedy algorithm is to add one more read or *contig* (set of overlapping reads) to any given read or contig, until no more operations are possible. The contigs made by greedy extension have the highest scoring overlap, based on the used scoring function (each software implementation has a different scoring function). The greedy algorithms are implicit graph algorithms, where only the highest-scoring edges are considered. Local maxima and false-positive overlap may cause problems to these algorithms. Software packages belonging to this class are SSAKE, SHARCGS and VCAKE.

2.2.2. Overlap Layout Consensus (OLC)

OLC algorithms are based on an overlap graph, a particular graph where the vertices represent reads and the edges represent overlaps. The overlaps must be precomputed with pairwise alignments, with a series of computational expensive operations. The basic idea of this method is that any paths through the graph are a potential targets. OLC approach was developed for previous generations assemblers, but some softwares extend this approach to work with NGS data. The OLC assembly has three main phases: the first one is overlap discovery, an all-against-all pair-wise read comparison. Often heuristics are used in this phase, in order to reduce complexity. The second phase is construct and manage the overlap graph, with information given by previous phase. The last phase is a multiple sequence alignment (MSA) in order to determine the target. Unfortunately, no efficient method for optimal MSA exist at the moment. Softwares belonging to this class are Edena, Shorty, Newbler and CABOG.

2.2.3. De Bruijn Graph (DBG)

DBG algorithms are based on de Bruijn graph, graphs where the edges correspond to a subpart of the reads and the vertices correspond to overlaps between these parts. Each read induces a path and reads with overlap induce a common path, without any pair-wise precomputation. For this reason, the DBG approach is one of the most frequently used for NGS data. Graph construction often employs a constant-time hash table lookup for the existence of each common subsequence. The problem of repetitions in target induces cycles in the graph and these allow to obtain more than one possible target. To reduce sensitivity to data sequencing error often a strong preprocessing phase is provided. Software products belonging to this class are Euler, Velvet, ABySS, AllPaths and SOAPdenovo.

2.2.4. Maximum likelihood genome assembly

The basic assumption in all genome assembly algorithms is that the goal of assembly process is to find the assembled genome with minimum length. The known problem of repeats in target DNA cause that these repeats can be under-represented in the shortest genome find by classic algorithms. Medvedev and Brundo in [7] suggest that *"the overall goal of an assembler should be not to minimize the length of the assembled genome, but to maximize the likelihood that it was the source of the set of reads"*. They note that high coverage of NGS data make possible to statistically estimate the read copy count (number of read's appearance in genome). Their assembly problem formulation is to maximize the likelihood of observed read frequencies rather than minimizing the length of the genome. They solve the new problem on a bidirected de Bruijn graph like a minimum cost bidirected flow problem with convex cost.

The framework suggested in [7] is to find genome that maximizes the global read count likelihood, but the framework is based on some assumptions. They use a circular genome D of length $N(D)$ and a red set of n reads of length k. Let d_i the number of times that read i appears in D, for a given i the probability that the outcome of a single trial is i is $\frac{d_i}{N(D)}$. Let the random variable X_i denote the number of trials whose outcome is i. All these 4^k variables are considered independently and following a binomial distribution. Their joint distribution is:

$$P(X_1 = x_i, X_2 = x_2, ..., X_{4^k} = x_{4^4}) = \frac{n!}{\prod x_i} \prod_i \left(\frac{d_i}{N(D)} \right)^{x_i}$$

D in not known while the results of the n trials are known. They consider the likelihood of the parameters of the distribution (d_i) given the outcome of the trials (x_i) and they call this the global read count likelihood:

$$L(d_1, ..., d_{4^k} | x_1, ..., x_{4^k}) = \frac{n!}{\prod x_i} \prod \left(\frac{d_i}{N(D)} \right)^{x_i}$$

The aim is to assemble the genome with the maximum global read-count likelihood. With a larger number of reads, the multinomial distribution can be approximated as a product of binomial distribution of each X_i (when number of reads goes to infinity, X_i become independent). Since in the binomial approximation $N(D)$ is a constant independent from each d_i, it is possible to replace $N(D)$ with N. N is the real length of actual genome from which the reads were sampled. Assuming that genome size is known, the resulting approximation for L is :

$$L(d_1, ..., d_{4^k} | x_1, ..., x_{4^k}) \approx \prod P(X_i = x_i) = \prod \binom{n}{x_i} \left(\frac{d_i}{N} \right)^{x_i} \left(1 - \frac{d_i}{N} \right)^{n - x_i}$$

The first experiments reveal high quality reconstruction of the genome, with NGS input data.

An improvement to [7] was presented by Varma, Ranade and Aluru in [10]. The idea is consider all possible genomes which contain all the reads that have been collected and pick the one which has the greatest probability of generating the observed read set. This approach eliminate previous knowledge of genome length and identical read length constraint. In this model each read r_i of length L_i is generated by a sequence G with length L_G, starting by a position uniformly chosen in $1 \ldots L_G - L_i$. Suppose a read r_i appears δ_i times in G, the probability of r_i being generated is $P(L_i)\delta_i/(L_G - L_i)$, where $P(L_i)$ denotes the probability of selecting a length L_i for the read. The probability of generating the read set $\mathcal{R}$ is $\prod_i P(L_i)\delta_i/(L_G - L_i)$. Since the same read set $\mathcal{R}$ can be generated through different permutations, the exact probability is $\alpha \prod_i P(L_i)\delta_i/(L_G - L_i)$, where α depends upon the read set $\mathcal{R}$ but not on the sequence G. Assuming $L_G \gg L_i$ it possible to write $L_G - L_i \approx L_G$ and so the probability P_G if generating $\mathcal{R}$ from G is :

$$P_G = \alpha' \prod_{i=1}^{n} \frac{\delta_i}{L_G}$$

The constant $\alpha' = \alpha \prod_i P(L_i)$ depends on $\mathcal{R}$ but not on G. The goal is to find the genome G that maximizes P_G. The experimental results demonstrate that the maximum likelihood approach has the potential to lead a big improvement in genome assemblers, particularly in terms of assembly quality. The major drawback of this approach is the increase of computational complexity. The development of a comprehensive assembler based on the maximum likelihood formulation is still an open issue.

2.2.5. Pro and cons of current approach

All these methods share some fundamentals elements. First of all, all methods use graph algorithms to achieve the goal. Each approach differs in the use and meaning associated with nodes, edges, and weights. Second, single base sequencing error induces false nodes/edges (false positive and false negative overlaps) that can induce a false path. Third, repeats in the target and sequencing error can produce branch and loop into the graph. In the graph context, assembly is a graph reduction problem and most of these reductions belong to NP-hard class. For this reason, assemblers use heuristics and approximation algorithms to manage error, remove redundancy, simplify the graph and reduce complexity. Where possible convex optimization techniques are adopted too.

Heuristics are fundamental for any commercial assembler: the success of an assembler depends largely on the sophistication of its heuristics for real reads including error, real genomes including repeats, and the limitations of modern computers. Though heuristics help to mitigate all these problems, attention to optimality, completeness, accuracy and precision of proposed solutions must be paid. Moreover, there is a lack of in-depth knowledge of current heuristics used in genome assembly. Often some parameters configuration are employed only because the experimental evidence shows their goodness, without any understanding of the underlying reason. Furthermore, heuristics are very sensible to the particular genome type.

Also the preprocessing phase is fundamental for a correct assembly. Some operations on read set, like eliminating low quality reads or trying to increase data reliability, are performed in all major assembler.

Between available methods, OLC and DBG are two robust approaches to assembly problem. OLC is better with longer read (100-800 bp range) and lower coverage, DBG is more appropriate for large amount of short read (25-100 bp range). In addition, DBG methods are also less computational expensive than OLC approaches.

On the other hand, the new maximum likelihood methods produce very quality assembly but are more computationally expensive than classic DBG methods. These methods are often used in post-processing, so their effectiveness is limited to output data of other classic methods.

Currently no algorithm nor software solve the whole genome sequencing assembly problem.

3. Our probabilistic approach

3.1. Introduction to the stochastic model

The introduction of maximum likelihood approach gives a new area of study for researchers in genome assembly and resurges research in shotgun assembly methods. Inspired by maximum likelihood approach, a new experimental probabilistic approach to de novo whole genome assembly has been developed. The stochastic model describes the main elements involved in sequencing, with the goal of finding a probabilistic relation between read set and genome sequence which generates it. With minor changes, the model can be modified to work with old generation data. The assumptions made by model are:

- reads are sequences of pairs (c, q) where c is a symbol and q is a quality value
- reads have a fixed length m
- read set has a fixed size M
- symbols belong to an alphabet Σ
- genome sequences have a fixed length N
- all genome sequences $\tilde{S} \in \Sigma^N$ are equiprobable

$$f_G(\tilde{S}) := P_G(S = \tilde{S}) = \left(\frac{1}{|\Sigma|}\right)^N \quad \tilde{S} \in \Sigma^N \tag{3.1}$$

- quality value q belongs to a set Q finite or countable
- symbol error probability uses the phred formulation. The error probability function can be defined as

$$f_e(q) := P_e(q) = 10^{-\frac{q}{10}} \quad \forall q \in Q \tag{3.2}$$

- with a sequence $\tilde{S}$ and reads of length m the probe positioning process starts from some position j in $\Omega_{pos} = [1, N - m + 1]$. The probability function for elementary events $(J = j)$ is

$$f_{pos}(j) := P_{pos}(J = i) \quad \forall j \in \Omega_{pos} \tag{3.3}$$

3.2. The basic idea

The read set is the output of a whole experiment performed by a sequencer. An interesting probability is the one that links the read set $\mathcal{R}$ to a particular sequence $\tilde{S} \in \Sigma^N$

$$f_{G|\mathcal{R}}(\tilde{S}, \mathcal{R}) := P_{G,\mathcal{R}}(S = \tilde{S}|T = \mathcal{R}) \tag{3.4}$$

The idea behind the model is that the unknown original sequence, from which the read set is produced, has the greatest probability among all the possible sequences. Without any other assumption, the only way to derive (3.4) is by Bayes' rule

$$P_{G,\mathcal{R}}(S = \tilde{S}|T = \mathcal{R}) = \frac{P_{G,\mathcal{R}}(S = \tilde{S})}{P_{G,\mathcal{R}}(T = \mathcal{R})} P_{G,\mathcal{R}}(T = \mathcal{R}|S = \tilde{S}) \tag{3.5}$$

Assuming that $P_{G,\mathcal{R}}(S = \tilde{S})$ (sequence probability) is constant and equal for all possible sequences and observing that also $P_{G,\mathcal{R}}(T = \mathcal{R})$ (the read set probability) is constant in the same experiment

$$P_{G,\mathcal{R}}(S = \tilde{S}|T = \mathcal{R}) \propto P_{G,\mathcal{R}}(T = \mathcal{R}|S = \tilde{S}) \tag{3.6}$$

Now the problem can be treated as a maximum likelihood problem.

3.2.1. Read set posterior probability and maximum likelihood principle

The maximum likelihood principle requires to find which is the probability of the evidence given the model and then to maximize over all the models in order to find the one which maximize the probability of the evidence. In this case the model is defined for the sequence $\tilde{S}$ and is mathematically described by the function f_G (3.1), while the evidence is given by the read set $\mathcal{R}$.

In order to derive a closed form for $P_{G,\mathcal{R}}(T = \mathcal{R}|S = \tilde{S})$ the following hypothesis has been introduced.

- **Read conditional independence**: two different reads of the same read set $\mathcal{R}$ are statistically independent given the reference sequence

Introducing read probability $f_{r|G}$, the probability formulation become

$$f_{\mathcal{R}|G}(\mathcal{R}|\tilde{S}) = \mu(\mathcal{R}) \prod_{h=1}^{M} f_{r|G}(\mathbf{r_h}, \mathbf{q_h}, \tilde{S}) \tag{3.7}$$

where $\mu(\mathcal{R})$ is the multinomial coefficient

$$\mu(\mathcal{R}) = \binom{M}{k_1 k_2 ... k_{\bar{M}}} \tag{3.8}$$

given that $\mathcal{R}$ contains $\bar{M}$ distinct reads and each read appears k_i times $(\sum^{\bar{M}} k_i = M)$. Now only a closed form for $f_{r|G}$ is needed to complete the model. In order to obtain $f_{r|G}$, some probabilities have been introduced.

3.2.2. Single pair probability

Sequencing machines produce a pair (c, q) with $c \in \Sigma$ and $q \in Q$ for each sequenced position. With finite set Σ and Q, it's only necessary to explicitly calculate the probability of elementary events $(C = c \cap Q = q)$ for all $(c, q) \in \Omega_C = \Sigma \times Q$. With hypothesis of independence between symbols and quality values and equiprobable symbols and quality values a closed form for single pair probability $f_C(c, q)$ is

$$f_C(c, q) := P_C(C = c \cap Q = q) = \frac{1}{|\Sigma||Q|} \tag{3.9}$$

Supposing to know that a pair (c, q) has been produced by sequencing the position k of the reference sequence S the probability that $S_k = \gamma$ given the evidence (c, q) for all possible $\gamma \in \Sigma$ is

$$f_{G|C}(\gamma, c, q; k) := P_{G,C}(S_k = \gamma | C = c \cap Q = q) \tag{3.10}$$

Note that k is a constant parameter of the model, so it can be omitted. To derive a closed form for (3.10) some hypothesis have been formulated.

- **Uniform error probability**: the error probability $P_e(q)$ for the pair (c, q) is uniformly distributed among all symbols $\gamma \neq c$

$$f_{G|C}(\gamma, c, q; k) := P_{G,C}(S_k = \gamma | C = c \cap Q = q) = \begin{cases} 1 - P_e(q) & \gamma = c \\ \frac{P_e(q)}{|\Sigma| - 1} & \gamma \neq c \end{cases} \tag{3.11}$$

- **Local sequencing information**: if pair (c, q) comes from position k of sequence S then $P_{G|C}(\gamma, c, q; i) = P_G(S_k = \gamma) \quad \forall i \neq k$

Previous assumptions are critical in the model. Using quality value in probability formula allows to weigh data reliability in assembly choices.

3.2.3. Single read probability

A positioned read is a triple $(\mathbf{r}, \boldsymbol{q}, j)$ in $\Omega_P = \Omega_C^m \times \Omega_{pos}$ where $\boldsymbol{r} = r_1 r_2 ... r_m \in \Sigma^m$, $\mathbf{q} = q_1 q_2 ... q_m \in Q^m$ and $j \in \Omega_{pos}$.

The probability of elementary events $(\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q} \cap J = j)$ is

$$f_P(\mathbf{r}, \mathbf{q}, j) := P_P(\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q} \cap J = j) \;\; \forall(\mathbf{r}, \boldsymbol{q}, j) \in \Omega_P \tag{3.12}$$

To derive a closed form of f_P it's necessary to introduce the following hypothesis:

- **Intraread symbol independence**: the process of producing the pair $(r_i,\, q_i)$ of the read $(\mathbf{r}, \mathbf{q})$ is statistically independent from the process of producing the pair $(r_j,\, q_j)$ of the same read for all $i \neq j$,

$$P_P(\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q}) = \prod_{l=1}^{m} P_C(R_l = r_l \cap Q_l = q_l) \tag{3.13}$$

- **Positioning sequencing independence**: the process of probe positioning is statistically independent from actual read produced.

$$P_P(\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q} \cap J = j) = P_C(\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q}) P_{pos}(J = j) \tag{3.14}$$

With both the hypothesis in conjunction the model will greatly simplify

$$f_P(\mathbf{r}, \mathbf{q}, j) = \prod_{l=1}^{m} P_C(R_l = r_l \cap Q_l = q_l) P_{pos}(J = j) = f_{pos}(j) \prod_{l=1}^{m} f_C(r_l,\, q_l) \tag{3.15}$$

Positioned read are an effective way to describe sequences of symbol-quality pairs. Unfortunately no prior information is available on actual sequencing position for the read. Sequencer produce just reads in the form $(\mathbf{r}, \mathbf{q})$ without any indication of their position j. With this observation, the elementary events is $\{\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q}\} = \{\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q} \cap J = \Omega_{pos}\}$. So the probability is

$$f_r(\mathbf{r}, \mathbf{q}) := P_r(\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q}) = \sum_{j \in \Omega_{pos}} f_P(\mathbf{r}, \mathbf{q}, j) \tag{3.16}$$

Introducing the following hypothesis:

- **Uniform positioning**: positioning process is uniform distributed along the sequence. So (3.3) become

$$f_{pos}(j) = P_{pos}(J = i) = \frac{1}{|\Omega_{pos}|} = \frac{1}{N - m + 1} \forall j \in \Omega_{pos} \tag{3.17}$$

and using the previous closed form for f_C (3.9), then (3.16) becomes

$$f_r(\mathbf{r}, \mathbf{q}) = \sum_{j \in \Omega_{pos}} f_P(\mathbf{r}, \mathbf{q}, j) = \sum_{j \in \Omega_{pos}} \left[f_{pos}(j) \prod_{l=1}^{m} f_C(r_l,\, q_l) \right] = \prod_{l=1}^{m} f_C(r_l,\, q_l) = \left(\frac{1}{|\Sigma||Q|} \right)^m \tag{3.18}$$

3.2.4. Positioned read posterior probability given the sequence

In order to derive a closed form for $f_{r|G}$, the positioned read posterior probability given the sequence has been introduced.

Considering a positioned read $(\mathbf{r}, \boldsymbol{q}, j)$ and a sequence $\tilde{S}$ the posterior probability is

$$f_{G|P}(\tilde{S}, \mathbf{r}, \mathbf{q}, j) := P_{G,P}(S = \tilde{S} | \mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q} \cap J = j) \quad (3.19)$$

To derive a closed form for (3.19) the following hypotheses have been introduced.

- **Read locality**: read $(\mathbf{r}, \boldsymbol{q})$ sequenced from position j of the reference sequence S "influence" only knowledge of S_k for $k = j, ..., j + m - 1$, where m is the length of the read.

$$f_{G|P}(\tilde{S}, \mathbf{r}, \mathbf{q}, j) = \begin{cases} P_{G,P}\left(\bigcap_{k=j}^{j+m-1}(S_k = \tilde{S}_k | \mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q} \cap J = j)\right) & j \leq k \leq j + m - 1 \quad (a) \\ P_{G,P}\left(\bigcap_{k<j,\, k \geq j+m}(S_k = \tilde{S}_k)\right) & otherwise \quad (b) \end{cases} \quad (3.20)$$

Now only for (3.20(a)) is needed to find a closed form, since for (3.20(b)) it is possible to use (3.1). To achieve this goal, other hypothesis has been used.

- **Position sequence independence**: the probe positioning process is statistically independent from the actual sequence

$$P(S = \tilde{S} \cap J = j) = P(S = \tilde{S}) \cap P(J = j)$$

With previous hypothesis and using some probability property, the (3.20(a)) becomes

$$f_{G|P}(\tilde{S}, \mathbf{r}, \mathbf{q}, j) = P_{G,P}(\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q} | S_{j,m} = \tilde{S}_{j,m}) \frac{P_{G,P}(S_{j,m} = \tilde{S}_{j,m}) P_{G,P}(J = j)}{P_{G,P}(\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q}) P_{G,P}(J = j)} \quad (3.21)$$

To derive a closed form for

$$P_{G,P}(\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q} | S_{j,m} = \tilde{S}_{j,m}) \quad (3.22)$$

the following hypotheses have been used.

- **Symbols conditional independence**: two different pairs of the same read are statistically independent given the reference sequence

$$P(\mathbf{R} = \mathbf{r} \cap \mathbf{Q} = \mathbf{q} | S = \tilde{S}) = \prod_{l=1}^{m} P(R_l = r_l \cap Q_l = q_l | S = \tilde{S}) \quad (3.23)$$

- **Local sequencing**: the production of the pair (r_l, q_l) from position j of the reference sequence is statistically independent from all symbols S_i for $i \neq j$

Using the last hypotheses and applying Bayes' rule is possible to obtain

$$f_{G|P}(\tilde{S},\mathbf{r},\mathbf{q},j)=\prod_{l=1}^{m}f_{G|C}(\tilde{S}_{j+l-1},r_l,q_l;j+l-1)\cdot\frac{\prod_{l=1}^{m}f_C(r_l,q_l)}{\prod_{l=1}^{m}f_G(\tilde{S}_{j+l-1})}\cdot\frac{f_G(\tilde{S}_{j,m})}{f_r(\mathbf{r},\mathbf{q})} \quad (3.24)$$

Taking into consideration the hypothesis in (3.18) and equiprobable symbols (3.24) the (3.20(a)) becomes

$$f_{G|P}(\tilde{S},\mathbf{r},\mathbf{q},j)=\prod_{l=1}^{m}f_{G|C}(\tilde{S}_{j+l-1},r_l,q_l;j+l-1) \quad (3.25)$$

3.2.5. Single read posterior probability given the sequence

To derive a closed form for $f_{r|G}$, consider (3.20)

$$\begin{aligned}&f_{P|G}(\mathbf{r},\mathbf{q},j,\tilde{S}):=P_{G,P}(\mathbf{R}=\mathbf{r}\cap\mathbf{Q}=\mathbf{q}|S_{j,m}=\tilde{S}_{j,m})\\&=\frac{P_{G,P}(S_{j,m}=\tilde{S}_{j,m}|\mathbf{R}=\mathbf{r}\cap\mathbf{Q}=\mathbf{q})P_{G,P}(\mathbf{R}=\mathbf{r}\cap\mathbf{Q}=\mathbf{q})}{P_{G,P}(S_{j,m}=\tilde{S}_{j,m})}\end{aligned} \quad (3.26)$$

Summing over all the possible positions to manage the lack of information on the position j

$$\begin{aligned}&f_{r|G}(\mathbf{r},\mathbf{q},\tilde{S}):=\sum_{j\in\Omega_{pos}}f_{pos}(j)f_{P|G}(\mathbf{r},\mathbf{q},j,\tilde{S})\\&=f_r(\mathbf{r},\mathbf{q})\sum_{j\in\Omega_{pos}}\frac{f_{pos}(j)}{f_G(\tilde{S}_{j,m})}f_{G|P}(\mathbf{r},\mathbf{q},j,\tilde{S})\\&=f_r(\mathbf{r},\mathbf{q})\sum_{j\in\Omega_{pos}}\frac{f_{pos}(j)}{f_G(\tilde{S}_{j,m})}\prod_{l=1}^{m}f_{G|C}(\tilde{S}_{j+l-1},r_l,q_l;j+l-1)\end{aligned} \quad (3.27)$$

Now a closed form for $f_{r|G}$ has been obtained.

3.2.6. The final formulation

Using previous formulas, a closed form for (3.7) is:

$$\begin{aligned}f_{\mathcal{R}|G}(\mathcal{R}|\tilde{S})&=\mu(\mathcal{R})\prod_{h=1}^{M}f_{r|G}(\mathbf{r_h},\mathbf{q_h},\tilde{S})\\&=\mu(\mathcal{R})\left(f_r(\mathbf{r},\mathbf{q})\frac{f_{pos}(j)}{f_G(\tilde{S}_{j,m})}\right)^M\prod_{h=1}^{M}\left[\sum_{j\in\Omega_{pos}}\prod_{l=1}^{m}f_{G|C}(\tilde{S}_{j+l-1},r_l,q_l;j+l-1)\right]\end{aligned} \quad (3.28)$$

The (3.28) can be used with different implementation of f_r, f_{pos}, f_G and $f_{G|C}$. The previously closed form are only a possible choices, used in model development.

However, using the proposed closed form a very simple formulation can be achieved.

- **The simplest case**: assuming that the sequence $\tilde{S}$ is defined as an i.i.d. process, with probability of a single symbol $|\Sigma|^{-}1$ (3.1). Supposing moreover that the positions are also described by and i.i.d. process like in (3.17) for all j and f_r is (3.18), then (3.27) becomes

$$f_{r|G}(\mathbf{r}, \mathbf{q}, \tilde{S}) = \frac{1}{(N-m+1)|Q|^m} \sum_{j \in \Omega_{pos}} \prod_{l=1}^{m} f_{G|C}(\tilde{S}_{j+l-1}, r_l, q_l; j+l-1) \quad (3.29)$$

With these assumption (3.28) becomes

$$\begin{aligned} f_{\mathcal{R}|G}(\mathcal{R}|\tilde{S}) &= \mu(\mathcal{R}) \prod_{h=1}^{M} f_{r|G}(\mathbf{r_h}, \mathbf{q_h}, \tilde{S}) \\ &= \mu(\mathcal{R}) \left(\frac{1}{(N-m+1)|Q|^m} \right)^M \prod_{h=1}^{M} \left[\sum_{j \in \Omega_{pos}} \prod_{l=1}^{m} f_{G|C}(\tilde{S}_{j+l-1}, r_l, q_l; j+l-1) \right] \end{aligned} \quad (3.30)$$

3.2.7. Read set probability

Sequencer produces millions of reads per run. A correct read set probability has to take into account all information coming from a single read. Defining the elementary events as $\mathcal{R} = \{(\mathbf{r}, \mathbf{q})_h : h = 1, ..., M\}$ the read set $\mathcal{R}$ probability function is

$$f_{\mathcal{R}}(\mathcal{R}) := P_{\mathcal{R}}(T = \mathcal{R}) \quad (3.31)$$

Deriving a closed form for $f_{\mathcal{R}}$ directly from f_r is not possible. Indeed the process of producing two different reads in the same read set is not statistically independent, since it is precisely the correlation between reads which makes possible the assembly.

The only way to reach a different formulation is by marginal distribution

$$P_{\mathcal{R}}(T = \mathcal{R}) = \sum_{\tilde{S} \in \Sigma^N} P_{\mathcal{R}|G}(T = \mathcal{R}|S = \tilde{S}) P_G(S = \tilde{S}) \quad (3.32)$$

Read set probability can be use to “measure” the reliability of a given read set: with same coverage, read length, read set size and original sequence a greater value for (3.32) suggests a well done read set (greater correlation between reads and probably better assembler results).

3.3. Comparison with current approaches

The stochastic model starts from maximum likelihood idea to create an entire probabilistic framework for genome assembly. Each component and each phase has been described in a probability manner. While in graph algorithms maximum likelihood is only used as a criterion in path choice and post-processing, in this approach is the fundamental of all assembly process. One of the main differences with current approaches is that this model proposes an exact approach, without heuristics. All current methods use heuristics to achieve the assembly goal, although heuristics are very sensible to many factors and their massive use can affect data correctness. Moreover the utilization of heuristic is often data dependent. In this approach, instead, there is a solid mathematical and stochastic base: all assembly elements are described with a precise model and stochastic process. Also it is an exhaustive approach, since all sequences are studied in order to identify likelihood. As main consequence, the model gives an exponential formulation of assembly problem. The current exhaustive approach, however, was not designed to be the final model's formulation, but it is only exploited to give a first model validation. Another important characteristic of the proposed implementation is the fundamental use of quality values. The majority of current methods do not use or poorly use quality values. Usually they employ them during a preprocessing phase, when reads with low quality values are removed. Instead, this approach strongly exploits quality values in assembly phase. Indeed (3.11), which is a core assumption in proposed model implementation, employ the quality value to weigh the positive or negative base matching. The quality value explains the reliability in current read-sequence base comparison and this information is used during likelihood computation. Moreover is not necessary to use a preprocessing phase, since low quality reads are managed by model. Another important feature is that all produced results are measurable: for all the sequences tested a probability/likelihood values are produced. So direct comparison between single experiment's results and, with some assumption, among different experiment's results can be achieved. Instead, in current approaches a comparison between results is very difficult. Another innovative feature is the information given by read set probability. The read set probability can be further used as an index for read set reliability, so multiple read set produced by same DNA can be directly compared.

4. The program development

4.1. Program goals

The development of the software follows the main purposes from which the model has been created.

The first main goal is to give a first implementation of the model, which has never been tested before this work. A first implementation of all probability functions can provide more information about feasibility, correctness and robustness of the model. Moreover, correctness of some used hypotheses and assumptions can be tested.

To achieve these objectives the software has been developed in a very modular way, to guarantee a flexible tool also for future exploitation. Each component of the stochastic model has been modeled with a data type and/or a function. Also the organization of source files guarantees a high modularity. For the main probabilities used in the model, different functions have been developed, each with different tradeoff between performance and flexibility.

The second and much important goal has been to create a test program, a simple assembler, to perform the first experiments with artificial data. The simulation's results can provide some interesting ideas for future changes or improvements to the model.

In order to achieve these goals an assembler has been developed to read an input read set and elaborate it. Read set probability (3.32) and sorted read set posterior probabilities (3.28) are the most important produced data.

Given the exhaustive approach used by this model and the exponential nature of DNA assembly problem, some optimizations to source code have been performed in order to limit execution time and reach greater input size.

A multithreading approach and the departmental IBM P770 Power7 have been used in development and test phases to support computational efforts.

4.2. The stochastic model into the software

All the previous exposed model has been implemented in the software: all the probabilities listed in previous chapter have a counterpart as program function. Each function receives in input the same parameters of the probability function, to ensure an intuitive and bijective map between formulas and functions.

With the double aim of having a debuggable version and providing some ready to use functions, a closed form of all probabilities in model has been implemented into the software. These hypotheses have been added to model assumptions (sec. 3.1) in order to produce the first runnable version.

The hypotheses used for this version are:

- equiprobable quality value elements
- equiprobable symbols (3.1)
- independence between symbol and quality value and equiprobable symbols and quality values (3.9)
- uniform positioning (3.17)
- single read probability closed form (3.18)
- uniform error probability (3.11)
- positioned read posterior probability closed form (3.25)
- single read posterior probability closed form (3.29)
- read set posterior probability (3.32)

4.3. Challenges, implementation choices and platform

The main challenge coming from the model definition is its exponential formulation. The proposed exhaustive approach to the assembly problem requires an accurate analysis in software development and platform choice, to mitigate exponential issues. Another main challenge is to guarantee the necessary numeric precision at computed probability value. As consequence of exponential problem nature, single event can have very small probability values. These two challenges have been addressed during first phase of software development.

4.3.1. Exponential problem

Trying to reduce side effects of exponential problem nature, some elements have been analyzed. First of all, the model formulation exposes an embarrassing parallel chance. Indeed, the entire set of 4^N sequences to elaborate can be divided in subsets

and the computation can be executed in a parallel way, since no relation between sequences has been described in the model. So the available departmental IBM P770 Power7 seemed to be the perfect choice. This system has all the hardware and software requirements for application's development: a custom OS, a high performance compiler and some parallel tools. Among all the available languages, the C language has been chosen for its relative "low-level" feature and the possible benefit arising from installed compiler. Moreover, the shared memory system's feature and the large amount of available memory suggest a multithreading approach to parallelism. Using C language, a Pthread approach has been adopted for multithreading. In the following sections, all these elements are exposed in detail.

4.3.1.1. The C Programming Language

In order to develop the program, the C language has been chosen. C is an imperative, general-purposes and relative "low-level" programming language. The main reason of this choice is the relative "low-level" feature, which can offer better performance than a high-level language if properly exploited. Obviously an extra workload and a correct use of this feature is required. On the other hand, the exponential problem's nature and the exhaustive approach needs all possible aid from programming language. The C language adoption appears one of the best possible choices.

In program development, a C99 standard has been adopted. This choice has been guided by the great diffusion of this standard and the increasing support to floating point introduced in this version. Moreover, no extra-libraries have been used in program development.

4.3.1.2. POSIX Pthread

To increase performance of test program and exploit IBM P770 Power7 potential, a multithreading approach has been adopted. Using C language, a classic and consolidated option for UNIX based system is POSIX Pthread. Pthread is a standardized C language threads programming interface, specified by the IEEE POSIX 1003.1c standard. In shared memory multiprocessor architectures threads can be used to implement parallelism and the POSIX standard guarantees the needed performance and portability features. Since the IBM P770 Power7 is a shared memory machine running an UNIX base OS, called AIX, Pthread seemed to be a good choice. Also a MPI approach has been considered in a preliminary analysis, but using a shared memory machine a multithread approach is better than a multiprocess solution. Indeed, while multiple threads in same application share the same address space, multiple processes have their own separate memory area.

4.3.1.3. IBM P770 Power7

All tests performed in this work have been performed on IBM P770 Power7. IBM P770 is a mid range system which includes up to 64 core POWER7. The used version has:

- 6 Power7 Processors (3 drawers, each with 2 IBM Power7 Processors)
- 640 GB RAM
- 16 TB hard disk space (external SAN storage)
- AIX 6.1

IBM P770 is a shared memory machine, so each core can access to all available memory. This feature has been very important for the multithreading approach adopted on software development.

POWER7 is a superscalar symmetric multiprocessor, based on Power Architecture. and released in 2010 as successor of POWER6. Compared to its predecessor, POWER7 has an increase power efficiency through multiple cores and simultaneous multithreading (SMT). Main POWER7 features are:

- RISC architecture
- 45 nm technology, $1,2 \cdot 10^9$ transistors
- 8 Cores per processor
- 4 way SMT (4 threads/core)
- 32KB + 32 KB L1 cache/core
- 256 KB L2 cache/core
- 32 MB L3 cache/processor, on chip
- Memory controller on chip

4.3.1.4. AIX, LoadLeveler and XL C compiler

IBM P770 operative system is AIX 6.1, a proprietary UNIX base OS developed by IBM. Since its first version in 1986, AIX is standard OS for many company's computer platforms. Using an IBM OS into a IBM systems is often the best choice. Also, AIX has an own implementation of Pthread library, conforming to IEEE POSIX 1003.1c standard.

On AIX was installed LoadLeveler, the IBM's batch scheduling system. IBM LoadLeveler is a very effective tool for managing parallel and serial jobs, matching job requirements with the machine resources. User submits a job using a job command file and the LoadLeveler scheduler attempts to find resources to satisfy the requirements of the job. Moreover, LoadLeveler tries to maximize the efficiency of the whole system, maximizing the utilization of resources and minimizing the job turnaround time.

Also, on AIX was installed XL C compiler, the standard IBM C compiler with fully support to ISO/IEC 9899:1999 (C99) and ISO/IEC 9899:1990 (referred to as C89). XL C is an advanced and high-performance compiler, designed to optimize and tune applications for execution on IBM Power platforms. Installed and used version is 11.1, which introduce support for POWER 7 processor.

4.3.2. Numeric precision

The second challenge has been to guarantee the necessary numeric precision in probability values computation. Probabilities are intrinsically small number, with value between 0 and 1. The exponential nature of assembly problem causes that simple events may have very small probability values. C standard type for floating point values are `float`, `double` and `long double`. The most precise type, the `long double` type, has a guarantee precision of about 15 digits. Introduced in C89 and improved in C99, `long double` may not necessarily map to an IEEE format, so implementation is architecture-dependent. Moreover, the declared precision is not uniform but vary along real axis. To partially overcome this limitation and achieve a better precision, a custom floating point data type and a complete arithmetic have been developed. Although this solution partially overcomes machine precision issues, a price has been paid in performance: standard C type better exploit hardware potential than custom types. However, the desire of sufficient precision in probability computation has forced this choice.

4.3.2.1. Custom floating point data type

In design and implementation, the data type structure took inspiration from IEEE 754 standard. A mantissa and an exponent have been specified, adding a exponentiation base and a number for digits in mantissa.

```
/*
        Floating point number in scientific notation.
        Mantissa (man) , exponent (exp) ,
        base (base) and number of not decimal digits (digit)
        X = man * base^exp
*/
typedef struct sciDouble{
        long double man;
        int exp;
        int base;
        int digit;
} sciDouble, t_longdouble;
```

The developed functions include cast functions (from `long double` to `t_longdouble` and reverse), comparison functions and arithmetic functions. In arithmetic functions, hybrid parameters have been allowed because software provides an automatic cast to custom type. Arithmetic functions include: addition, subtraction, multiplication and division.

```
/* Multiplication between two custom type*/
sciDouble mul_s_s(const sciDouble op1, const sciDouble op2);

/* Multiplication between a long double and a custom type*/
sciDouble mul_d_s(const long double op1, const sciDouble op2);

/* Multiplication between a custom type and a long double*/
sciDouble mul_s_d(const sciDouble op1, const long double op2);

/* Division between two custom type*/
sciDouble div_s_s(const sciDouble op1, const sciDouble op2);

/* Division between a long double and a custom type*/
sciDouble div_d_s(const long double op1, const sciDouble op2);

/* Division between a custom type and a long double*/
sciDouble div_s_d(const sciDouble op1, const long double op2);

/* Addition between two custom type*/
sciDouble add_s_s(const sciDouble op1, const sciDouble op2);

/* Addition between a long double and a custom type*/
sciDouble add_d_s(const long double op1, const sciDouble op2);

/* Addition between a custom type and a long double*/
sciDouble add_s_d(const sciDouble op1, const long double op2);

/* Subtraction between two custom type*/
sciDouble sub_s_s(const sciDouble op1, const sciDouble op2);

/* Subtraction between a long double and a custom type*/
sciDouble sub_d_s(const long double op1, const sciDouble op2);

/* Subtraction between a custom type and a long double*/
sciDouble sub_s_d(const sciDouble op1, const long double op2);
```

Multiplication has been performed adding the two exponents and multiplying the two mantissa. Then, the number of digits in result mantissa were normalized to specified number.

```
/*
        Multiplication between two sciDouble
        The result has the same digit and the same base of first factor
*/
sciDouble mul_s_s(const sciDouble op1, const sciDouble op2){

        sciDouble result;

        ...

        /*calculate mantissa*/
        result.man = op1.man * op2.man;

        /*calculate exponent*/
        result.exp = op1.exp + op2.exp;

        /*set the correct digit number*/
        result = changeDigit(result, op1.digit);

        return result;
}
```

Similarly, division has been performed subtracting the two exponents, dividing the two mantissa and normalizing digits in result mantissa. For both operations, control on exponent base and on number of digits has been performed.

Addition and subtraction have been implemented executing algebraic sum of mantissa, for addends with the same exponent. If the exponents are different, previously the smaller addend is brought to same exponent of greater one. As obvious, control on exponent base and number of digits have been designed.

Even though not used in final software, also exponentiation and logarithm have been developed for possible future uses.

```
sciDouble logWithBase_s(sciDouble op, int base);
sciDouble logBase_s(sciDouble op);
sciDouble log2_s(sciDouble op);
sciDouble log10_s(sciDouble op);
sciDouble pow_s(sciDouble op, int expo);
```

4.3.2.2. Formulas simplification

In order to both reduce precision issues, both to increase performance, some changes have been done on core functions that implements (3.29).

The original (3.29) formula was:

$$f_{r|G}(\mathbf{r},\mathbf{q},\tilde{S}) = \frac{1}{(N-m+1)|Q|^m} \sum_{j\in\Omega_{pos}} \prod_{l=1}^{m} f_{G|C}(\tilde{S}_{j+l-1}, r_l, q_l; j+l-1)$$

The $|Q|^m$ factor on fraction's denominator causes a further rapid decrease of probability value when used in (3.30). Since it is a constant value and for ranking purposes was irrelevant (it does not alter the relative order) in the final version of the software it has been omitted. As a consequence, the calculated value is not anymore a probability but is an index which differs from the real probability by a constant factor.

$$f_{r|G}(\mathbf{r},\mathbf{q},\tilde{S}) \propto \frac{1}{(N-m+1)} \sum_{j\in\Omega_{pos}} \prod_{l=1}^{m} f_{G|C}(\tilde{S}_{j+l-1}, r_l, q_l; j+l-1) \tag{4.1}$$

Hence also (3.30) which used this function, now returns an index value, no more a probability value.

Always with the aim of mitigating the limited precision issues, a Stirling approximation has been introduced in factorial computation.

$$\mu(\mathcal{R}) = \binom{M}{k_1 k_2 \ldots k_{\bar{M}}} = \frac{M!}{k_1!k_2!\ldots k_{\bar{M}}!} \simeq (\tfrac{M}{e})^M \sqrt{2\pi M} \frac{1}{k_1!k_2!\ldots k_{\bar{M}}!}$$

The $M!$ factor at multinomial coefficient's numerator quickly exceeds machine precision. Using Stearling approximation in (3.30) allows to spread operation into multiplication. Adding (4.1) simplification, the final formula is:

$$f_{\mathcal{R}|G}(\mathcal{R}|\tilde{S}) \propto \frac{\sqrt{2\pi M}}{k_1!k_2!\ldots k_{\bar{M}}!} \prod_{h=1}^{M} \left[\frac{M}{(N-m+1)e} \sum_{j\in\Omega_{pos}} \prod_{l=1}^{m} f_{G|C}(\tilde{S}_{j+l-1}, r_l, q_l; j+l-1) \right] \tag{4.2}$$

For DNA assembly purposes, an index directly proportional to real probability value has the same utility. This solution allows to mitigate some problems in elaboration complexity and numeric precision.

4.4. The assembler - Description and pseudocode

The assembler uses the probability functions which have been developed in a first phase of the work. The program output is a complete ranking of all input sequences, sorted by descending probability. Program also computes multiplication between sequence probability and the probability sum of all elaborated sequences, which corresponds to read set probability if given input sequences are Σ^N. The assembler takes in input the read set file (in simil-FASTA format), the list of sequences to elaborate and the quality value range.

Algorithm 4.1 Assembler

read input read set file
create read set object
read input sequences file
create sequence list
calculate and **initialize** parameters ***//(N, M, m, number of sequence, quality value range)***
create and **initialize** thread parameters for probability computation
assign sequence list sub-part to each thread
for each thread
    start thread for probability computation ***//parallel execution***
wait threads for probability computation
update read set probability $P(R)$
create and **initialize** thread parameters for merge operation
assign results sub-part to each thread
for each thread
    start thread for merge operation ***//parallel execution***
wait threads for merge
print ranking
print $P(R)$

4.4.1. Input and parameters setting

The first operations done has been the elaboration of the read set, reading input file and creating a `ReadSet` type struct. `ReadSet` contains and array of `Read` struct type. Each read is a sequence of `Pair` data type.

Then, the input sequence and quality value range are read and all parameters are set (sequence size N, read set size M, read length m, sequences number $seqNum$, minimum quality value Q_min and maximum quality value Q_MAX). Also, read set multinomial coefficient is calculated, using a previously written function.

4.4.2. Probability computation

The most important part of the program is probability computation. For each input sequence the read set posterior probability index is calculated, using function (4.2). Also, all calculated probabilities are accumulated in a variables. If input sequences are Σ^N, then multiplying this variable for sequence probability in main, the read set probability index can be obtained.

In the developed program, each thread takes a subset of all sequences and for each sequences in its subset calculates read set posterior probability index (4.2). Let NS the number of input sequence and NT the number of thread, then each thread has about $NL = \frac{NS}{NT}$ sequences to elaborate. Moreover, each thread sorts its NL sequences by probability in ascending order with `qsort` library function.

Algorithm 4.2 Thread probability computation

thread_probability(ReadSet, SequenceListSubPart, ProbabilityList)
  P(ReadSet)_local = 0 ***//local read set probability***
  for each sequence S_i **in** SequenceListSubPart
    ProbabilityList[i] = **calculate** $P(ReadSet|S_i)$ ***//read set posterior probability***
    P(ReadSet)_local = P(ReadSet)_local + $P(ReadSet|S_i)$
  sort ProbabilityList
  return P(ReadSet)_local

4.4.3. Ranking

The last part of the program performs the DNA sequences ranking, sorting input sequences by just calculating probability values. Since a local sorting of each thread's data has already been performed , after probability computation the result array has NT sorted subparts. So, merging these NT sorted subparts, a complete sequences sorting is obtained. In order to exploit multithreading, a parallel approach has

been applied to the merging phase. Each thread merges a subset of sequences and writes the merged sequences in a common array, at the right place. So the first thread merges the smallest elements, the second thread merges the next ones, and so on until last thread that merges the greatest ones. First of all, the subset of sequences to assign to each thread must be identified. To achieve this goal, starting by an observation: the greatest value merged by *i-th* thread must be not greater than any elements merged by *i+1-th* thread, since each thread merges subsets of increasing values. Considering that all threads have to merge elements in different array subparts, for each thread and for each sorted array subpart a range of indices must be found. Let $x[j]$ be the *j-th* element in first sorted data set. In this subpart, all elements $x[j]$ at position $j = i \cdot \frac{NL}{NT}$, with $i \in \{1, ..., NT-1\}$, are found. Then, with a binary search in all other $NT-1$ array subparts, the indices of the same elements $x[j]$ (or the greater smaller elements) in any other subpart are found. Now each thread has its own ranges of indices where applying merge procedure.

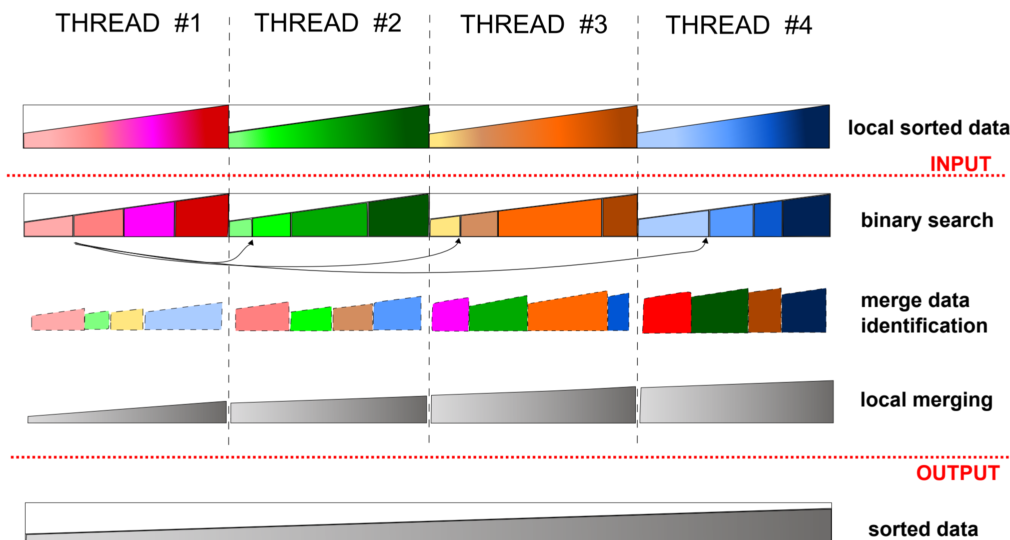


Figure 4.1.: Merge multithread with NT=4 threads

Each thread merges NT sorted sub-part and writes sorted results into a common output array. Every thread creates a local array of NT entries, in order to store current index of each sub-array. Then until there are no more data to be merged, first elements in each NT assigned sub-parts are examined to find minimum. At each iteration, every thread adds the local minimum at the correct position on common result array.

Algorithm 4.3 Thread merge operation

```
thread_merge(InputResultsList, OutputResultList, MergeIndicesRange, ThreadNumber)
    currentIndexArray[ThreadNumber]    //current index for each sub array
    i_result_start = 0    //start index for output array
    N_local = 0    //amount of data to merge for this thread
    min_tmp = +∞    //temporary minimum
    i_min //index of current minimum
    for each sub array
        initialize currentIndexArray
    initialize i_result_start and N_local
    for (i=0; i < N_local ; ++i )
        min_tmp = +∞
        for (j=0; j < ThreadNumber; ++j)    //find min
            if( currentIndexArray[j] ∈ MergeIndicesRange AND
              InputResultsList[currentIndexArray[j]] < min_tmp )
                min_tmp = InputResultsList[currentIndexArray[j]]
                i_min = j
        OutputResultList[i_result_start+i] = min_tmp    //add min value
        update currentIndexArray[i_min]
    return
```

4.4.4. Output

The program produces in output the complete or partial ranking and the read set probability value, if input sequences are Σ^N. Additional information like reads overview, total execution time and average time per sequence are provided.

4.5. Code optimization solutions

4.5.1. General best practices adopted

Although programming style is very personal, there are programming practices that will help to get the best results from the optimization techniques used by the compiler. In code development, some IBM tips and guidelines have been adopted to exploit the C for AIX compiler potential. Some guidelines adopted where possible are:

- Pass a value as an argument to a function, rather than letting the function take the value from a global variable.
- Use constant arguments in functions, to provide more opportunities for optimization.
- Fully prototype all functions. A full prototype gives the compiler and optimizer complete information about the types of the parameters. As a result, promotions from unwidened types to widened types are not required, and parameters can be passed in appropriate registers.
- Design functions so that they have few parameters and the most frequently used parameters are in the leftmost positions in the function prototype.
- Avoid passing by value structures or unions as function parameters or returning a structure or a union. Passing such aggregates requires the compiler to copy and store many values. Instead, pass or return a pointer to the structure or union, or pass it by reference.
- If your function exits by returning the value of another function with the same parameters that were passed to your function, put the parameters in the same order in the function prototypes. The compiler can then branch directly to the other function.
- Use the built-in functions, which include string manipulation, floating-point, and trigonometric functions, instead of coding your own. Intrinsic functions require less overhead and are faster than a function call, and often allow the compiler to perform better optimization.
- In a structure, declare the largest members first.
- In a structure, place variables near each other if they are frequently used together.
- Use local variables, preferably automatic variables, as much as possible. The compiler must make several worst-case assumptions about global variables.
- Use constants instead of variables where possible. The optimizer is able to do a better job reducing runtime calculations by doing them at compile time instead.

- Use register-sized integers (long data type) for scalars.
- Avoid forcing the compiler to convert numbers between integer and floating-point internal representations.
- Keep array index expressions as simple as possible.
- Use pre-increment/decrement in loop indices

4.5.2. Specific code optimization

Some extra optimizations have been done in core assembler's parts. Very strong optimizations have been applied only on functions used in the final version of assembler, in order to increase program performance.

A very important component in the program is the custom floating point arithmetic, since all probability computations consist of some real numbers operations. The `t_longdouble` data type and the developed arithmetic functions have been strongly optimized, reducing input parameters controls and reorganizing arithmetic operations.

In *priorprobability_c.h*, has been created an array with the precalculated sequence probabilities. Since equiprobable symbols hypothesis has been used, precalculating the sequence probability is possible and allow to reduce computation.

In *posteriorprobability_c.h*, have been created two arrays containing the precalculated values for (3.11). Since the single pair posterior probability has been used in comparison between each sequence's symbols and each reads pairs, it is a crucial component for assembler. For example, testing all sequences with N=14 on a read set with m=4, M=35 about $4^N \cdot M \cdot m = 35 \cdot 4^{15} = 37'580'963'840$ comparison were done and each would result in a division, an addition and an exponentiation. Since with hypothesis of finite symbols alphabet and finite quality values set is allowed to list all possible values, this solution guarantees an important speedup.

4.6. Compiler optimization and tuning

The compiler optimization and tuning can significantly improve program performance, only specifying some flag in compiling procedure. XL C compiler offers a large number of optimization and tuning options. First of all, to correctly compile multithread application the right compiler invocation is necessary. In order to create threaded applications and link programs that use multithreading, XL C provides compiler invocations with a suffix _r (xlc_r) to specified thread-safe compilation. In the following list, the flags used in makefile are listed:

- *-O5*: Most aggressive optimizations available (reorganization or elimination of global data structures, loop optimizations, Interprocedural analysis (IPA), etc.)
- *-q64*: Generates code for a 64-bit addressing model (64-bit execution mode)
- *-qarch=pwr7*: Generate instructions that are optimized for a POWER7 architecture.
- *-qsimd=auto*: Automatically take advantage of vector instructions for processors that support them, if possible.
- *-qlargepage*: Takes advantage of large pages provided on POWER4 and higher systems.
- *-qsmp=auto*: Enables automatic parallelization of program code, if possible.

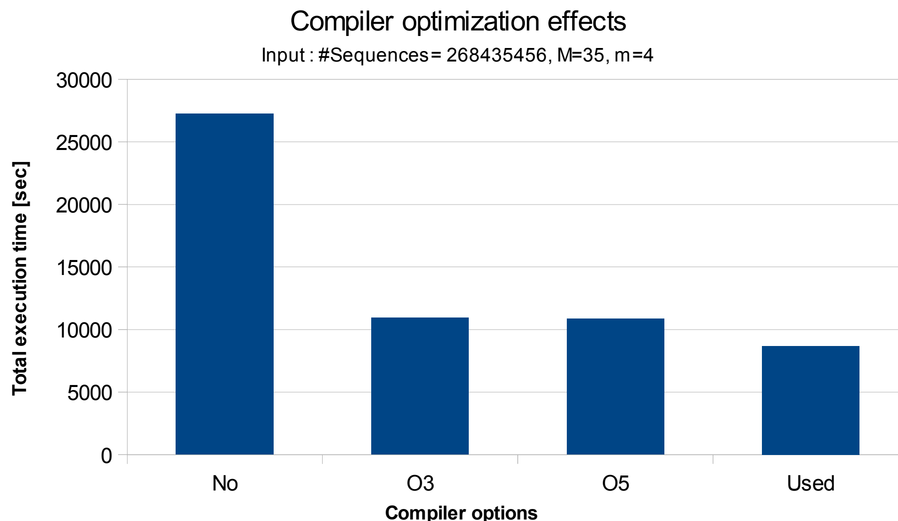


Figure 4.2.: Compiler optimization

Using an O3 optimization level, execution time is about 2.5 times lower than without any optimization flag. Instead, the difference between O3 and O5 optimization levels is very small. Introducing even previous listed flags, the execution time is reduced by about 20% than an O5 optimization level only.

5. Simulations and experiments

5.1. Modus operandi

All read sets used in the test phase are composed of artificial data. A python script has been used to generate the reads as sequences of pairs [base; quality values], from a randomly generated input sequence. The script receives as input a sequence of size N, a quality value range [Q_min,QMAX], a read length (m) and a read set size (M). The script, with a random equiprobable positioning process between 1 and $N - m + 1$, creates the reads. First, it starts reading m consecutive bases from the current position, then it adds random quality values to each base. This operation was repeated M times for each generated read. In order to make the experiment more realistic, some random errors have been introduced in the created read sets: in pairs with low quality values, some base were randomly changed. All tests have been performed with quality values in the range [10;35], so there are very few simulated errors.

Four types of tests have been performed in order to validate:

- the ranking feature correctness
- the ranking feature reliability
- the ranking feature robustness
- the read set probability index

The experiments have been performed with the version of the software introduced in *chapter 4*, using 48 threads and the most performant compiler optimization level. In all the tests, the input configurations have been chosen to not result in a run time of the program greater than 12 hours. This choice has been adopted in order to both not spend too time on the machine and allow a greater tests repeatability.

5.2. Ranking

5.2.1. Correctness

In these tests, the main goal is to study the correctness of the ranking feature. Different test configurations have been tested varying coverage (values 3, 5 and 10), in order to observe the model behavior in different situations.

Coverage 3 A coverage 3 is a small value, especially in a NGS context, but it has been employed anyway to study model behavior.

Table 5.1.: Test Ranking (TOP 20): C≃3.14; N=14; M=11; m=4; 0 base errors

Rank	Sequence	Value	Hamming Dist. and differences
1°	ATACGTTCACTGCG*	$653010323403034 \cdot 10^{-19}$	0
2°	ATACGTTCACTGCA	$549019953290166 \cdot 10^{-21}$	1 (last base)
3°	ATACGTTCACTGCT	$548444737077953 \cdot 10^{-21}$	1 (last base)
4°	ATACGTTCACTGCC	$548440765170250 \cdot 10^{-21}$	1 (last base)
5°	GTACGTTCACTGCG	$467215874570474 \cdot 10^{-21}$	1 (first base)
6°	TTACGTTCACTGCG	$464077448482833 \cdot 10^{-21}$	1 (first base)
7°	CTACGTTCACTGCG	$464072087708780 \cdot 10^{-21}$	1 (first base)
8°	ATACGTTCACTGTG	$143689673906261 \cdot 10^{-21}$	1 (second-last)
9°	ATACGTTCACTGAG	$143631054079042 \cdot 10^{-21}$	1 (second-last)
10°	ATACGTTCACTGGG	$143587025516808 \cdot 10^{-21}$	1 (second-last)
11°	CAATACGTTCACTG	$113471576180316 \cdot 10^{-21}$	right shift
12°	ATATACGTTCACTG	$112504504772884 \cdot 10^{-21}$	right shift
13°	ATACGTTCACACTG	$110435173685824 \cdot 10^{-21}$	2
14°	ACATACGTTCACTG	$109974928666097 \cdot 10^{-21}$	right shift
15°	ATACGTGTTCACTG	$109850753661478 \cdot 10^{-21}$	5
16°	ATACGTTCACTGTA	$109647827363929 \cdot 10^{-21}$	2 (last two)
17°	ATACGTTCACTGTC	$109646302540338 \cdot 10^{-21}$	2 (last two)
18°	ATACGTTCACTGTT	$109646256394306 \cdot 10^{-21}$	2 (last two)
19°	ATACGTTCACTGAA	$109625959777839 \cdot 10^{-21}$	2 (last two)
20°	ATACGTTCACTGAC	$109625573192179 \cdot 10^{-21}$	2 (last two)

Execution time ≃ 48 min (10,718 μs per sequence) *=reference sequence

The true sequence is the first in the ranking, so this time the program has found the correct sequence. The next 6 sequences differ from the true sequence in either the last or first base, while the next 3 sequences differ from the true one in the second-last base. The last five sequences in the TOP 20 ranking differ by the reference sequence on the last two bases. In the middle part of the ranking (positions 11, 12 and 14), instead, some sequences are partial shifts of the original sequence: a large subpart of

the original sequence appears in the test sequence as well, but in the wrong position. All these observations suggest that the most likely sequences differ from the reference sequence at the extremes. This may suggest that the sequence's extremes need more coverage. Another observation is that the sequence extreme reads, in a linear DNA contest, contribute intrinsically to a lesser extent in the assembly phase than the other reads. Indeed these reads overlap only on one side with other reads, bringing less information to the assembly process. As a consequence, the original sequence's centre "CGTTCAC" is the only part that appears in all TOP 20 sequences.

Table 5.2.: Test Ranking (TOP 10): C=3.2; N=15; M=12; m=4; 1 base errors

Rank	Sequence	Value
1°	TCATGCCATAGCTGC	$220152081209690 \cdot 10^{-21}$
2°	TCATAGCTGCCATGC	$220152081209690 \cdot 10^{-21}$
3°	AGCTGCATGCCATGG	$206856831096433 \cdot 10^{-21}$
4°	AGCTGCCATGCATGG*	$206856831096433 \cdot 10^{-21}$
5°	ATGCAGCTGCCATGG	$530829823226891 \cdot 10^{-22}$
6°	AGCTGCCATGGATGC	$519720035346666 \cdot 10^{-22}$
7°	ATGGAGCTGCCATGC	$517146903810176 \cdot 10^{-22}$
8°	ATGCCATGGAGCTGC	$517146903810171 \cdot 10^{-22}$
9°	CCATCATGCAGCTGG	$173901133450613 \cdot 10^{-22}$
10°	CCATGCAGCTCATGG	$119861950730758 \cdot 10^{-22}$

Execution time $\simeq$ 230 min (12,798 μs per sequence) *=reference sequence

In this simulation, the true sequence is fourth in the rank. The used read set has one simulated error in covering position [9,12]: the produced DNA fragment was "TCAT" instead of "GCAT". From the results it appears that the first and second sequences used this wrong piece of information during the assembly process. In low coverage read sets, the sequencing error can affect results correctness. Moreover, the original sequence has some repetitions inside, that does not help the assembly process. A difference between this simulation and the previous one is in the sequence probability index values: in the first experiment the gap between first and second sequence indices is greater than in the last simulation. This fact may suggest that when lower quality read sets (or with low coverage) are used, the difference between first position and second position is very small.

Coverage 5 A coverage 5 is a medium value for NGS data. Some experiments on sequences with lengths 12, 13, 14 and 15 have been performed.

Table 5.3.: Test Ranking (TOP 10): C=5.3; N=15; M=20; m=4; 0 base errors

Rank	Sequence	Value	Hamming Dist. and differences
1°	ATCTAGCACTTCACG*	$166485089598584 \cdot 10^{-18}$	0
2°	ATCTAGCACTTCACT	$711683176850489 \cdot 10^{-21}$	1 (last base)
3°	ATCTAGCACTTCACA	$889956328748221 \cdot 10^{-22}$	1 (last base)
4°	ATCTAGCACTTCACC	$889954752844185 \cdot 10^{-22}$	1 (last base)
5°	CATCTAGCACTTCAC	$449583305703572 \cdot 10^{-22}$	right shift 1
6°	TATCTAGCACTTCAC	$448664222767083 \cdot 10^{-22}$	right shift 1
7°	AATCTAGCACTTCAC	$448641077538416 \cdot 10^{-22}$	right shift 1
8°	GATCTAGCACTTCAC	$448639732796229 \cdot 10^{-22}$	right shift 1
9°	ATCTTCACTCTAGCA	$182396651698028 \cdot 10^{-23}$	10
10°	TCTAGCATCTTCACT	$168244040919235 \cdot 10^{-23}$	left shift

Execution time $\simeq$ 348 min (19,427 μs per sequence) *=reference sequence

In all tests with the same configuration of (Tab. 5.3) the reference sequence has been always on top of the rank. In the reported case, the second, third and fourth sequences in rank differ by the correct sequence in the last base only. Indeed, in the used read set there are only 2 reads that cover last part of the sequence: one covering positions [10,13] and another covering positions [11,14], so the last base is covered only by one read. On the other hand, sequences from position 5 to 8 are right shifts of original sequence. This fact can suggest that the assembly process is very sensitive at the extremes of the sequence, since only the core part “TCTAGCACTTCAC” appears in all the firsts 8 positions. The only common subsequence shared by all TOP 10 sequence is “CTTCAC”, that is completely covered by 4 reads in the read set (the 25% of total reads).

Table 5.4.: Test Ranking (TOP 10): C=5.3; N=15; M=20; m=4; 3 base errors

Rank	Sequence	Value
1°	CCTGCTTGATACCTA*	$328905325151445 \cdot 10^{-24}$
2°	CCTGCTTGATATACC	$129249699236084 \cdot 10^{-24}$
3°	ATACCTGCTTGATAT	$127988036353119 \cdot 10^{-24}$
4°	GATATACCTGCTTGA	$124405084559469 \cdot 10^{-24}$
5°	CTGCTTGATACCTAT	$562148723910261 \cdot 10^{-25}$
6°	TCTTGATACCTGCTT	$198298678027545 \cdot 10^{-25}$
7°	TCTGCTTGATACCTA	$637257750220209 \cdot 10^{-26}$
8°	GATACCTGCTTGATT	$577100070116333 \cdot 10^{-26}$
9°	GCTGCTTGATACCTA	$503904713491795 \cdot 10^{-26}$
10°	CTGCTTGATACCTAC	$498805137977172 \cdot 10^{-26}$

Execution time $\simeq$ 357 min (19,934 μs per sequence) *=reference sequence

In (Tab. 5.4) the same configuration has been tested, but with a read set containing 3 wrong reads (with one base error per reads). In this case, the difference between the first true sequence and the next ones is lower than in the previous test (similar to Tab. 5.2). Also, the errors in the read set do not affect the classification in the ranking, but they only affect the relative difference between subsequent values in the ranking.

With same coverage (5) but sequence length N=14, the same results have been achieved: the reference sequence is always on top of the rank, with reduced differences between subsequent values when the read sets contain errors.

Also with N=13 and N=12, almost all the times the true sequence has been identified. Two interesting simulations, where reference sequence has not been on top of rank, are showed.

Table 5.5.: Test Ranking (TOP 10): C=5.23; N=13; M=17; m=4; 2 base errors

Rank	**Sequence**	**Value**	**Hamming Dist. and differences**
1°	ATCTGCTAGGCTA	$455555683625862 \cdot 10^{-21}$	left shift 1
2°	ATCTGCTTGGCTA	$138118867659341 \cdot 10^{-21}$	1 from first seq.
3°	TATCTGCTAGGCT	$772546528110800 \cdot 10^{-22}$	1 (first base)
4°	ATCTGCTAGGCTG	$584117921830767 \cdot 10^{-22}$	1 from first seq.
5°	ATCTGCTAGGCTT	$581746951201724 \cdot 10^{-22}$	1 from first seq.
6°	ATCTGCTAGGCTC	$581668009707775 \cdot 10^{-22}$	1 from first seq.
7°	CATCTGCTAGGCT*	$570502332346344 \cdot 10^{-22}$	0
8°	GATCTGCTAGGCT	$569706022045535 \cdot 10^{-22}$	1 (first base)
9°	AATCTGCTAGGCT	$569553667484609 \cdot 10^{-22}$	1 (first base)
10°	ATCTGGCTATGCT	$442211711422233 \cdot 10^{-23}$	8 from first seq.

Execution time $\simeq$ 17 min (14,790 μs per sequence) *=reference sequence

In this simulation (Tab. 5.5) the true sequence is in position 7. The sequences in positions 1, 4, 5 and 6 are left shifts of the true sequence. This happens because the used read set does not contain reads covering position 0. Also, the sequences in positions 3, 8 and 9 differ by the correct sequence only in first base. Again, the sequence's extremes undercoverage causes problems in the assembly process. On the other hand, the 2 base errors are corrected by the assembly (wrong bases are in position 3 and 8 in the reference sequence) in all sequences until the 9th position, except for the second sequence.

Table 5.6.: Test Ranking (TOP 10): C=5; N=12; M=15; m=4; 1 base errors

Rank	**Sequence**	**Value**	**Hamming Dist. and differences**
1°	TCGAGAAAACAG	$819420742045327 \cdot 10^{-19}$	4 (last bases)
2°	TCGAGAAACAGA	$776866539943380 \cdot 10^{-19}$	1 (last base)
3°	ATCGAGAAACAG	$776769788055368 \cdot 10^{-19}$	right shift
4°	TCGAGAAACAGC	$776405514334398 \cdot 10^{-19}$	1 (last base)
5°	TCGAGAAACAGG*	$776342109463739 \cdot 10^{-19}$	0
6°	TCGAGAAACAGT	$776340485835887 \cdot 10^{-19}$	1 (last base)
7°	CTCGAGAAACAG	$776340484161624 \cdot 10^{-19}$	right shift
8°	TTCGAGAAACAG	$776340319934353 \cdot 10^{-19}$	right shift
9°	GTCGAGAAACAG	$776340214873981 \cdot 10^{-19}$	right shift
10°	TCGAAACAGAGA	$969175091478824 \cdot 10^{-22}$	4

Execution time $\simeq$ 4 min (12,417 μs per sequence) *=reference sequence

Here (Tab. 5.6) the true sequence is in position 5, 3 sequences differ in last base from the true sequence and other 4 sequences are right shifts of the original one. An observation of the employed read set reveals that the used data have not reads covering last 4 positions. As a consequence, the first sequence differs from the true one just in the last 4 bases. Again, the undercoverage in sequence's extremes may affect the ranking correctness.

Coverage 10 Coverage 10 is a good value for sequencer output. All simulations performed with this coverage value found the reference sequence, for all tested sequence length (N=12, 13, 14, 15).

Table 5.7.: Test Ranking (TOP 10): C≃10.66; N=15; M=40; m=4; 1 base errors

Rank	Sequence	Value	Hamming Dist. and differences
1°	CGACGGAACCTATCG*	$153895988814832 \cdot 10^{-12}$	0
2°	CGGAACCTATCGACG	$514579471875232 \cdot 10^{-17}$	11
3°	AACCTATCGACGGAA	$311313038766365 \cdot 10^{-17}$	left shift
4°	CGACGGAACCTATCC	$357466458609094 \cdot 10^{-18}$	1 (last base)
5°	CGACGGAACCTATCT	$349439913003182 \cdot 10^{-18}$	1 (last base)
6°	CGACGGAACCTATCA	$349059698543556 \cdot 10^{-18}$	1 (last base)
7°	AGACGGAACCTATCG	$965254997267898 \cdot 10^{-20}$	1 (first base)
8°	GGACGGAACCTATCG	$964128272309232 \cdot 10^{-20}$	1 (first base)
9°	TGACGGAACCTATCG	$955038485190722 \cdot 10^{-20}$	1 (first base)
10°	CCTATCGACGGAACC	$304351567846088 \cdot 10^{-20}$	left shift

Execution time ≃ 11.5 h (38,525 μs per sequence) *=reference sequence

In this example (Tab. 5.7), the correct sequence is on top of the ranking with a greater distance on the following sequences than in previous simulations. Although all sequences that differ in first or last base are still present in the rank, the higher coverage helps to mitigate the low amount of information provided by the sequence's extremes. The main consequence is that the gap between the first sequence and the second sequence is the greatest achieved so far.

Table 5.8.: Test Ranking (TOP 10): C≃10; N=14; M=35; m=4; 0 base errors

Rank	Sequence	Value	Hamming Dist. and differences
1°	TGCAGCCACAAACA*	$238353478452097 \cdot 10^{-12}$	0
2°	TGCAGCCACAACAA	$117419537618413 \cdot 10^{-14}$	3 (last bases)
3°	TGCAGCCACAACAG	$301564676385761 \cdot 10^{-15}$	3 (last bases)
4°	TGCAGCCACAACAC	$300219228987079 \cdot 10^{-15}$	3 (last bases)
5°	CTGCAGCCACAACA	$295838062275230 \cdot 10^{-15}$	9
6°	TGCAGCCACAACAT	$290918712762484 \cdot 10^{-15}$	3 (last bases)
7°	ATGCAGCCACAACA	$290578637262343 \cdot 10^{-15}$	9
8°	GTGCAGCCACAACA	$290575835575153 \cdot 10^{-15}$	9
9°	TTGCAGCCACAACA	$290575746987259 \cdot 10^{-15}$	8
10°	AGCAGCCACAAACA	$156212325537284 \cdot 10^{-19}$	1 (first base)

Execution time ≃ 145 min (32,322 μs per sequence) *=reference sequence

Also in this simulation (Tab. 5.8) the reference sequence is the first one, with a significant difference from the second placed one. The core part “CAGCCACAA” of the original sequence appears in all the other sequences in the ranking. So the sequence’s extremes uncertainty is still present but the high coverage allow to effectively manage this uncertainty.

5.2.2. Reliability

All the experiments with a coverage greater than 10 have always placed the reference sequence in first ranking position, also with read sets that contains limited errors in bases call.

On the other hand, the results have been more influenced by single errors in bases call when low coverage read sets have been employed. In order to study ranking feature reliability with low coverage, some tests have been performed multiple times with same parameters (quality values range, coverage, read length, read set size and original sequence) so correct sequence position's trend has been reported.

In the following test, from the random generated sequence "CGTAATAAGAGGCC" have been obtained 50 different read sets each of them with 18 reads of length 4 (so a coverage of about 5). The artificial data have single base quality values in range [10;35], so about 1 or 2 base error calls appear in each read set.

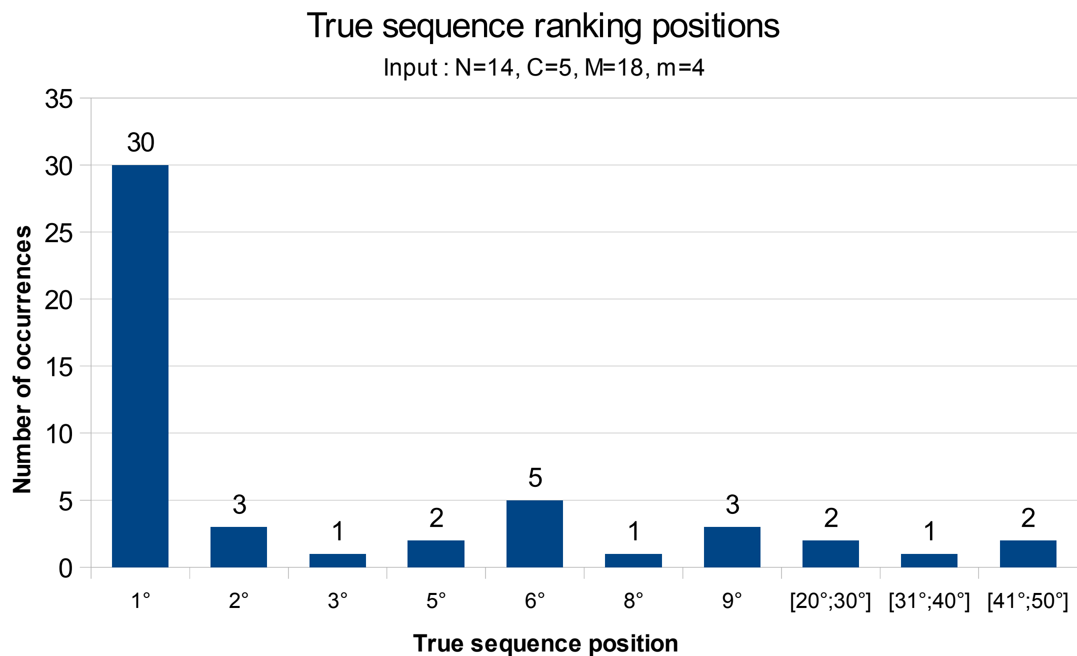


Figure 5.1.: True sequence ranking positions

In this test the 60% of rankings have been reported the true sequence in first position, the 30% until the 10th position and only 10% beyond the 20th position. However, the tests where the true sequences have been placed until the 10th position have sequence probability indices very close to first ones. Like in the previously described experiments, this simulation suggests that with low coverage the true sequence is almost always on top of the ranking, but when this does not happen the true sequence

is still at the top positions of the ranking. Therefore, in low coverage situations the reliability of the sequence probability index is very good but not error-free.

5.2.3. Robustness

In order to study the ranking feature robustness, some tests have been performed with low quality reads sets. To achieve this goal some read sets have been generated by a common DNA sequence, with same coverage and read length. Then an increasing number of reads in these read sets have been replaced by other randomly generated reads. Experiments have been performed with the double aim of checking true sequence ranking position and studying difference between the true sequence probability index and the first sequence probability index.

Coverage 5 and 20 reads The following tests have been performed with 7 read sets made by 20 reads with length 4. The original DNA sequence is "ATCTAG-CACTTCACG", so it has been achieved a coverage of about 5.

Table 5.9.: Test Ranking: C≃5; N=15; M=20; m=4

Number of replaced reads	True sequence position	First sequence probability index	First sequence	Second (or true) sequence probability index
0	1	$166485089598584 \cdot 10^{-18}$	ATCTAGCACTTCACG*	$711683176850489 \cdot 10^{-21}$
2	1	$130025478245528 \cdot 10^{-24}$	ATCTAGCACTTCACG*	$532569417000409 \cdot 10^{-27}$
4	6	$423231058802517 \cdot 10^{-35}$	GATCTAGCACTTCAC	$346713544796518 \cdot 10^{-38}$
6	62	$263609358967611 \cdot 10^{-39}$	ATCTAGGACTTCACT	$176798290454779 \cdot 10^{-44}$
8	38	$190223485016126 \cdot 10^{-45}$	GATCTAGCCTTCACT	$133101870467548 \cdot 10^{-48}$
10	13	$651931695616080 \cdot 10^{-51}$	GGTCTAGCACTCACG	$298543408548179 \cdot 10^{-52}$
15	>1000	$343437091870725 \cdot 10^{-57}$	ATGGACCACTAGTAG	-

Execution time $\simeq$ 47 h (about 400 min per read set) *=reference sequence

The test results show that the sequence probability index robustness suffers with low coverage. Indeed, only with few replaced reads the true sequence appears in first rank position.

Coverage 20 and 65 reads The following tests have been performed with 12 read sets made by 65 reads with length 4. The original DNA sequence is "ATTCCATG-GCTAC", so it has been achieved a coverage of 20.

Table 5.10.: Test Ranking: C=20; N=13; M=65; m=4

Number of replaced reads	True sequence position	First sequence probability index	First sequence	Second (or true) sequence probability index
0	1	$321569892716111 \cdot 10^{2}$	ATTCCATGGCTAC*	$746725166691417 \cdot 10^{-8}$
1	1	$249280792512051 \cdot 10^{2}$	ATTCCATGGCTAC*	$215738315600564 \cdot 10^{-8}$
2	1	$189480465113243 \cdot 10^{-7}$	ATTCCATGGCTAC*	$718865691461380 \cdot 10^{-22}$
3	1	$113334587210873 \cdot 10^{-7}$	ATTCCATGGCTAC*	$794780186670168 \cdot 10^{-23}$
4	1	$167371872781332 \cdot 10^{-13}$	ATTCCATGGCTAC*	$445539243568485 \cdot 10^{-18}$
5	1	$468113638691487 \cdot 10^{-17}$	ATTCCATGGCTAC*	$328003926307432 \cdot 10^{-31}$
10	1	$110166345321771 \cdot 10^{-32}$	ATTCCATGGCTAC*	$229622051781690 \cdot 10^{-41}$
15	1	$664401755792464 \cdot 10^{-43}$	ATTCCATGGCTAC*	$686901415898753 \cdot 10^{-56}$
20	1	$109099001904050 \cdot 10^{-61}$	ATTCCATGGCTAC*	$519173576919022 \cdot 10^{-68}$
30	2	$239413830061336 \cdot 10^{-80}$	ATTCCATGGCTAA	$312234991017309 \cdot 10^{-82}$
40	9	$305545113348599 \cdot 10^{-133}$	GTTCATGGCTACG	$721691630809830 \cdot 10^{-140}$
50	>1000	$268859671733745 \cdot 10^{-138}$	ATGGCTACCGTTC	-

Execution time $\simeq$ 15.5 h (about 78 min per read set) *=reference sequence

Until the number of replaced reads has been lower than 20 (about the 30% of reads), the true sequence has been always on top of the ranking. When a read set with 30 replaced reads has been used, true sequence dropped into second place, but with a small distance from the first sequence. Similar happens with 40 replaced reads, where the true sequence is in position 9. On the other hand, with 50 replaced reads the true sequence did not appear in TOP 1000 ranking. This test suggests that the probability sequence index is a robust index also with a large number of low quality data (replaced reads). With a high coverage, the first position in rank and a significant difference between subsequent values has been achieved also with corrupted data.

5.3. Read set

The following tests have been performed to study potential use of read set probability index as a reliability index for sequencing data. The base idea is that a more reliable read set has a higher probability value than a low quality read set. In order to perform these tests, some decreasing quality read sets have been generated, all with same coverages and read lengths. Then, an increasing number of reads in these read sets have been replaced by other randomly generated reads. Finally, a relation between read set probability index and the number of replaced reads has been analyzed.

Coverage 5 and 20 reads Given a sequence with length 15, 7 read sets made by 20 reads with length 4 have been created. Except the first one, in other read sets some reads have been replaced.

Table 5.11.: Test Read Set: C≃5; N=15; M=20; m=4

Number of replaced reads	Read set probability index
0	$156060326640470 \cdot 10^{-27}$
2	$122138948627109 \cdot 10^{-33}$
4	$398871546691510 \cdot 10^{-44}$
6	$297549798805752 \cdot 10^{-48}$
8	$134493061880969 \cdot 10^{-53}$
10	$344371728538061 \cdot 10^{-59}$
15	$526296309793931 \cdot 10^{-66}$

Execution time ≃ 47 h (about 400 min per read set)

The test results show a significant decrease in the read set probability index value with increasing replaced reads. Also with only few replaced reads, the difference between a real read set and a modified one is considerable.

Coverage 20 and 65 reads Similar tests have been performed with 12 read sets made by 65 reads with length 4. The original DNA sequence length is 13, so it has been achieved a coverage of about 20.

Table 5.12.: Test Read Set: C=20; N=13; M=65; m=4

Number of replaced reads	**Read set probability index**
0	$479176480874324 \cdot 10^{-6}$
1	$371457327291075 \cdot 10^{-6}$
2	$282347895373771 \cdot 10^{-15}$
3	$168881695286741 \cdot 10^{-15}$
4	$249410166281825 \cdot 10^{-21}$
5	$697543678718049 \cdot 10^{-25}$
10	$164160647332277 \cdot 10^{-40}$
15	$990035766054191 \cdot 10^{-51}$
20	$162570281055017 \cdot 10^{-69}$
30	$361407092261077 \cdot 10^{-88}$
40	$880836537765865 \cdot 10^{-141}$
50	$967735746078773 \cdot 10^{-146}$

Execution time $\simeq$ 15.5 h (about 78 min per read set)

In this test, the read set probability index value significantly decreases when many reads are replaced. The results seem to confirm the validity of the read set probability index as a reliability index for sequencing data. The decreasing rate of the index is not constant among the number of replaced reads: with only 2 replaced reads the index values is much larger than with 1 replaced read, while is comparable with the value for 3 replaced reads. This sensibility with a low number of replaced reads can be explained by the particular read employed for replacing. The chosen read, in fact, can be accidentally good for the assembly.

6. Conclusion and future work

This thesis presents a first experimental validation of a new maximum likelihood approach to the de novo genome assembly problem. The idea behind the model is that the unknown original DNA sequence, from which the read set is produced, has the greatest probability among all the possible DNA sequences. So elaborating all the sequences, it is possible to rank them based on the computed probability. The key idea is that the higher in the rank a sequence appears, the more likely it is the reference sequence. Moreover, the probability of the read set may offer some information about sequencer's data reliability: with the same coverage, read length, read set size and original sequence a greater value for this probability may suggests a good read set (greater correlation between reads and probably a better assembler results). Compared with the current methods, this model has an exhaustive approach to the assembly problem avoiding to employ heuristics, a strong use of quality values, a production of numerical measurable results and a solid mathematical and stochastic formulation of the assembly problem. On the other hand, both the proposed exhaustive approach and the exponential nature of the assembly problem require many resources both in elaboration time and in computational power. The current exhaustive approach, however, was not designed to be the final model's formulation but it is only exploited to give a the first model validation.

As explained in chapter 4, in order to give a first implementation of the model and to perform some experiments, a multithreading software assembler has been developed in the C language. A parallel approach using POSIX Pthread and the availability of an IBM P770 machine allow both to develop a high performance application for test purposes and to partially mitigate exponential issues. Furthermore, with the aim to reach the necessary numerical precision, a software implementation of the floating point type and a complete arithmetic have been developed. Also, strong code optimizations and some simplifications of probability formulas have been done in order to reduce execution time. The entire development phase has been characterized by find a balancing between performance and numerical precision.

The results of the experiments, reported in chapter 5, shows some interesting features. The ranking feature based on model assumptions is a reliable tool for DNA assembly. With a high coverage (greater than 10) and a limited number of base errors, the correct sequence is always at the top of the ranking with a significant gap between subsequent values. Even introducing a limited number (until 30%) of corrupted reads in the read set, the ranking feature remains accurate. So the sequence probability is a correct, reliable and robust index for a high coverage read

set, typical of the NGS context. With a lower coverage (5) the reference sequence is still often on top of the rank but with a reduced differences between subsequent values when the read sets contain errors. Using a coverage of 5, the reliability of ranking is very sensitive to the quality of the input read set. With a coverage value of 3, the same observations for coverage 5 remain valid but the sensitiveness in the read set quality is increased. A common observation, among all the coverage tested, is that assembly process correctness and reliability are significantly affected by the undercoverage at the extremes of the sequence. Often, only the core part of the reference sequence appears in all the sequences in the ranking, in the correct position or in a shifted position. In low coverage context, this fact may affect ranking correctness while with a high coverage the sequence's extremes uncertainty is effectively managed. Moreover, experiments show that the read set probability is a good index for the reliability of the input sequencing data. All simulations report a decrease of index value when many reads are corrupted, with all coverage employed.

In conclusion, the proposed maximum likelihood approach guarantees a high quality assembly for NGS-like data. Moreover, the stochastic model provides some useful indices than can be employ to analyze results correctness and reliability. For data with low coverage, the current formulation guarantees a very good but not error-free assembly. On the other hand, the main drawbacks are that the current exhaustive approach adopted by the model and the exponential nature of the assembly problem limit the size of input that can be elaborated in a reasonable time.

In a future development, the first problem that might be addressed will be to reduce the program execution time. This goal can be achieved through many ways. First of all, an algorithmic approach can be attempted. Studying the problem's formulation and the model's description, it can be possible to reach a reorganization of the computation that reduces the amount of processing. Indeed, having to elaborate all the 4^N sequences for a given sequence length N, maybe a clever utilization of the common sequence parts elaboration can achieve a good result. For example, an approach similar to the parallel prefix computation idea can be considered. Another direction worth exploring concerns the analysis of a different approach to the limited numeric precision. The current custom data type guarantees the desired precision but it increases the amount of computation. Maybe a different solution, closer to the hardware level, can be taken into account. Another idea to improve the program's performance is to switch from the current CPU based platform to a GPU based one. Indeed a GPU architecture, that executes a larger amount of simple operations than a CPU architecture, may be exploited for the program purposes. The types of the operations performed in the program (a large amount of simple floating point operations) may fit well with the GPU platform features. Obviously, all the previous ideas are only possible solutions to the problem, but none of them guarantee a sure improvement.

When some solutions to the execution time issues will be found, the second goal that might be addressed will be to compare the maximum likelihood approach with the current approaches. An in-depth comparison of the methods and the employment of

real data can provide both a further validation for the model, both some new ideas to improve the model formulation. Moreover, further tests can better explained some phenomena encountered during the preliminary tests, like the issues in the sequence's extremes undercoverage.

In summary, this thesis lays out the first groundwork for the developing of an assembler based on the maximum likelihood approach. Hopefully, further research will expand and improve it even more.

A. Source code organization

A.1. Data type

- *pair.h pair.c*: define `Pair` data type as a couple of quality value and nucleotide base. Defines functions to compare base, quality value and `Pair`.
- *read.h read.c*: define `Read` data type as an array of `Pair` and a read size. Defines functions to compare `Read` and print basic information.
- *readset.h readset.c*: define `ReadSet` data type as an array of `Read` and a read set size.
- *scientificdouble.h scientificdouble.c*: define `t_longdouble` data type as a mantissa, an exponent, an exponent base and a number of digit in mantissa. Defines functions to calculate addition, subtraction, multiplication, division, exponentiation, logarithm. Define functions to cast between C standard `long double` and `t_longdouble`, to compare two `t_longdouble`, to change exponent base and number of digits in mantissa.
- *t_result_data.h t_result_data.c*: define `t_result_data` data type as a pair of probability value and DNA sequence. Define function to compare two `t_result_data`.
- *threaddata.h*: define `thread_data` data type, used to packing arguments for multithreading probability function.

A.2. Utilities

- *constants.h*: define some shared constants used by application.
- *multinomialcoefficients_c.h multinomialcoefficients_c.c*: define functions to calculate factorial, multinomial coefficient denominator and complete multinomial coefficient.
- *sort.h sort.c*: define `t_merge_mt_data`, used to packing arguments for multithreading merge function. Define functions to binary search and multithreading merge.

- *probability_c.h probability_c.c*: define variable for sequence number and read set probability. Define functions to read and create read set, given an FASTA-like input file. Define function to read input sequences file. Define function to calculate read set multinomial coefficient (3.8) and read set multinomial coefficient denominator.

A.3. Probability function

- *readsetprobability_c.h readsetprobability_c.c*: define functions to calculate read set probability given a `ReadSet` data type (3.32).
- *posteriorprobabilitygivenevidence_c.h posteriorprobabilitygivenevidence_c.c*: define functions to calculate posterior probability given evidence. Give and implementation for (3.7) (3.27), (3.28), (3.30), (4.1), (4.2).
- *posteriorprobability_c.h posteriorprobability_c.c*: define functions to calculate posterior probability. Give an implementation for (3.4), (3.11), (3.20), (3.24), (3.25), (3.29).
- *priorprobability_c.h priorprobability_c.c*: define functions to calculate prior probability. Give an implementation for (3.1), (3.2), (3.3), (3.9), (3.15), (3.17), (3.18).

A.4. Assembler

- *testProbabilityPOSIX_c.c*: main program, receive in input sequences and read set, calculate probability and produce on output a sequence ranking and other index.

B. POSIX Pthread functions

The Pthreads API contains around 100 subroutines. The subroutines which comprise the Pthreads API can be informally grouped into four major groups: thread management, mutexes, condition variables and synchronizations. Only function in first group was used in software development. In this appendix, a brief description of main Pthread function used is done.

B.1. Thread creation

The main() program comprises a single, default thread. All other threads must be explicitly created by the programmer, with Pthread API.

To create a thread and make it executable there is function `pthread_create`, which can be called any number of times from anywhere within code.

Syntax

```
#include <pthread.h>
int pthread_create (
        pthread_t *thread,
        const pthread_attr_t *attr,
        void **start_routine (void),
        void *arg,
);
```

Parameters

- `thread`: unique identifier for the new thread returned by the subroutine.
- `attr`: attribute object that may be used to set thread attributes.
- `start_routine`: the C routine that the thread will execute once it is created
- `arg`: a single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void.

Return value

- If successful, the function returns zero. Otherwise, an error number is returned to indicate the error.

Error codes

The `pthread_create` function will fail if:

- EAGAIN: the limit on the number of threads in the class may have been met.
- EINVAL: the value specified by attr is invalid.
- EPERM: the caller does not have the appropriate permission to set the required scheduling parameters or scheduling policy.

B.2. Thread attribute

By default, a thread is created with certain attributes but some of these attributes can be changed via the thread attribute object. Attributes include: detached or joinable state, scheduling inheritance, scheduling policy, scheduling parameters, scheduling contention scope, stack size, stack address and stack guard (overflow) size.

Functions `pthread_attr_init` and `pthread_attr_destroy` are used to initialize/destroy the thread attribute object.

B.3. Thread termination

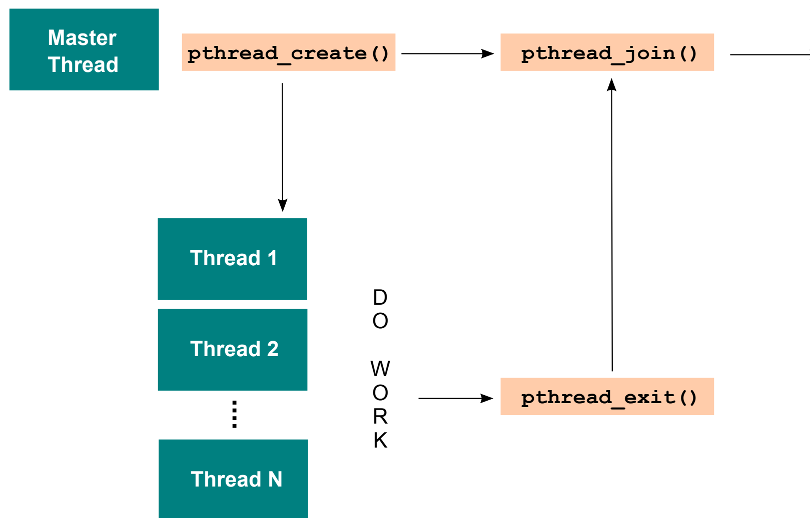
There are several ways in which a thread may be terminated:

- The thread returns normally from its starting routine. It's work is done.
- The thread makes a call to the `pthread_exit` subroutine - whether its work is done or not.
- The thread is canceled by another thread via the `pthread_cancel` routine.
- The entire process is terminated due to making a call to either the `exec()` or `exit()`
- If `main()` finishes first, without calling `pthread_exit` explicitly itself

The `pthread_exit()` routine allows to specify an optional termination status parameter. This optional parameter is typically returned to threads "joining" the terminated thread.

B.4. Thread join

Joining is one way to achieve synchronization between threads.



The `pthread_join()` subroutine blocks the calling thread until thread specified by `thread_id` terminates. The target thread's termination return status may be specified in the target thread's call to `pthread_exit()`. A joining thread can match one `pthread_join()` call only.

Syntax

```
#include <pthread.h>
int pthread_join(
        pthread_t thread_id,
        void **thread_return
);
```

Parameters

- `thread_id`: thread identifier of joining thread
- `thread_return`: pointer to location where store joining thread return value

Return value

- If successful, the `pthread_join()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

Error codes

The `pthread_join()` function shall fail if:

- EINVAL: the implementation has detected that the value specified by thread does not refer to a joinable thread.
- ESRCH: no thread could be found corresponding to that specified by the given thread ID.

Bibliography

[1] Ieee standard for binary floating-point arithmetic. *ANSI-IEEE Std 754-1985*, 1985.

[2] Blaise Barney. *https://computing.llnl.gov/tutorials/pthreads/ - POSIX Threads Programming*. Lawrence Livermore National Laboratory.

[3] IBM. *IBM XL C/C++ for AIX, V11.1 - Compiler Reference*. IBM Product documentation, April 2010.

[4] IBM. *IBM XL C/C++ for AIX, V11.1 - Language Reference*. IBM Product documentation, April 2010.

[5] IBM. *IBM XL C/C++ for AIX, V11.1 - Optimization and Programming Guide*. IBM Product documentation, April 2010.

[6] Inge Rodriguez Murali Paramasivam Keigo Matsubara, Edison Kwok. *Developing and Porting C and C++ Applications on AIX*. IBM Redbooks, March 2009.

[7] Paul Medvedev and Michael Brudno. Maximum likelihood genome assembly. *Journal of computational Biology*, 16(8):1101–1116, 2009.

[8] Jason R. Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315 – 327, 2010.

[9] Niranjan Nagarajan and Mihai Pop. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *Journal of computational biology*, 16(7):897–908, 2009.

[10] A. Varma, A. Ranade, and S. Aluru. An improved maximum likelihood formulation for accurate genome assembly. In *Computational Advances in Bio and Medical Sciences (ICCABS), 2011 IEEE 1st International Conference on*, pages 165–170, 2011.

Acknowledgments

Questa tesi è la conclusione di un percorso cominciato 5 anni fa, percorso nel quale molte persone mi hanno accompagnato e sento ora la necessità di ringraziare.

Ringrazio il Professor ***Gianfranco Bilardi***, relatore di questa tesi, per l'opportunità datami e per il supporto fornitomi nello svolgersi del lavoro. Spero di non aver deluso le sue aspettative e la fiducia in me riposta.

Ringrazio ***Michele Schimd***, mio compagno di viaggio in questa tesi e fonte di innumerevoli consigli e suggerimenti. Senza la tua guida questo lavoro non sarebbe stato lo stesso.

Il ringraziamento più grande va ai miei genitori ***Gianni*** e ***Alessandra*** che in tutti questi anni mi hanno supportato/sopportato, questa laurea è anche vostra. Senza il vostro affetto e la vostra forza non sarei qui oggi, vi voglio bene.

Un ringraziamento a tutti gli amici dell'università, se questi anni di studio sono volati è anche merito degli innumerevoli bei momenti passati insieme. Oggi e sempre, grazie.

Ringrazio in particolare ***All Dei Long*** (***Ale***, ***Bedo***, ***Colla***, ***Emma***, ***Fede***, ***Gio***, ***Marty***, ***Nico***, ***Vanuz***) per le serate, le vacanze, le pizze e tutti i bei momenti passati insieme. Siete stati una delle cose più belle di questa esperienza universitaria. Non vi dimenticherò.

Un grazie a ***Bedo*** e ***Giulio***, con i quali ho condiviso un fantastico viaggio in California nell'estate del 2011. E' stata un'esperienza meravigliosa e i miei compagni di viaggio sono stati parte del successo di questa avventura. Oltre ad avanzare ancora un pranzo di pesce...

Un grazie a tutti quelli che sono stati miei compagni di team, in particolare a ***Ale***, ***Bedo***, ***Colla***, ***Gio*** e ***Giulio***. Bei momenti e tanti successi, il PM è onorato di aver lavorato con voi.

Grazie al mio migliore amico ***Andrea***, che è stato sempre presente in tutti questi anni. Anche se non sai tirare le punizioni a PES, sei quasi come un fratello per me.

Grazie a ***Daniele***, ***Gabriele***, ***Luca***, ***Manuel***, ***Marco*** e ***Sebastiano***, già miei compagni alle superiori e matricole insieme a me ormai 5 anni fa. Tanto tempo fa l'impatto con l'università non è stato brusco anche per merito vostro.

Grazie a tutti i miei compagni di classe e ai docenti dell' ***ITIS Severi***, se l'informatica è ora parte della mia vita è anche per voi. Mille grazie per avermi accompagnato sulla mia strada.

Un ringraziamento a tutte le persone che mi hanno aiutato in un momento difficile, se ho raggiunto la maturità prima ancora di questa laurea è anche grazie a voi. La notte è sempre più buia subito prima dell'alba. A tutti, grazie.

PS: Un grazie infine ad Alfred, che mi fa trovare sempre il mantello stirato e la batmobile con il pieno. Numero uno!