Analysis and visualization of DNA sequences using cloud computing

Author:
Beat Wolf

Professor:
Prof. Pierre Kuonen

Client:
Dr. David Atlan, Phenosystems SA

Expert:
Dr. Peter E.M. Taschner

February 11, 2011
Abstract

While the research field of genetics is very popular these days and numerous research projects around the globe try to find new ways to approach the different problems present in this domain, not very much has been done to create an easy and comprehensible software suite that allows the user to do all the analysis of his data with one application.

During this project in collaboration with the enterprise Phenosystems SA, the different steps of the analysis and what tools currently exist to perform them were identified. The different tools were integrated into an application that allows the user without knowledge of the tools used to perform his analysis. Also different ways of how to use new technology to parallelize different resource intensive tasks were explored.

In the end a framework integrating different external and custom built tools was developed, and a visualization application developed as a semester project was improved and integrated into this framework, providing the user with the complete workflow from the moment the patient data was sequenced to the final analysis and diagnostic.

This project was developed in collaboration with the College of Engineering and Architecture of Fribourg, Phenosystems SA and two European research projects, Gen2Phen [2] and NMDChip [4]. It is a continuation of a semester project which can be found in Visualization of DNA sequence coverage, Beat Wolf, 10.06.2010, which can be found on the enclosed CD-ROM [A].

Acknowledgments

I would like to thank Dr. Pierre Kuonen that made my project possible and followed it with great interest. He was always available if I needed support.

I would like to specially thank Dr. David Atlan from Phenosystems that created this wonderful opportunity to work in this very interesting domain, genetics, which is in fact one of my personal fields of interest.

During this project, Valentin Clément, which developed POP-Java, was of great assistance and the collaboration with him was a pleasure.

I would also like to thank the European union “Seventh Framework Programme” [1] that helped financing the visits to two conferences I had the privilege to attend during this project. The NMDChip [4] conference in Würzburg and the Gen2Phen [2] conference in Montpelier. The grants IDs for those two projects are #223026 [5] and #200754 [3].

Lastly, I want to thank Dr. Peter E.M. Taschner from the Medical Center at the Leiden University, Netherlands, for his interest shown in this project.
Contents

1 Context ....................................................... 8

2 State of the art ........................................... 11

  2.1 Frameworks ............................................ 11

  2.1.1 GAPSS ................................................ 11
  2.1.2 Galaxy ............................................... 11

  2.2 Distributed tools ..................................... 12

  2.2.1 JCVI cloud BIO-Linux ............................... 12
  2.2.2 Cloudburst .......................................... 12
  2.2.3 Digipede ............................................ 12

  2.3 Specialized tools .................................... 12

  2.3.1 Samtools ........................................... 13
  2.3.2 Varscan ............................................. 13
  2.3.3 Bowtie .............................................. 13
  2.3.4 BWA .................................................. 13
  2.3.5 Stampy .............................................. 14

3 POP-Java .................................................. 15

4 Specification ............................................. 17

  4.1 Goals .................................................... 17

  4.1.1 Framework .......................................... 17
  4.1.2 Distributed computing integration ............... 17
  4.1.3 Visualization tool .................................. 18
  4.1.4 Usability ........................................... 18

5 Analysis .................................................. 19

  5.1 Reference scenario ................................... 19
  5.2 Parallelization ....................................... 21
  5.3 Use cases .............................................. 22

     5.3.1 Actors .............................................. 22
     5.3.2 Use case descriptions ............................ 23

6 Design ................................................... 25

  6.1 Deployment ............................................ 26
  6.2 Grid computing business model .................... 26
  6.3 Algorithms ............................................ 28

     6.3.1 Non-indexed alignment ............................ 28
     6.3.2 Indexed alignment ................................ 30
     6.3.3 Uniqueness map .................................. 32

7 Implementation ......................................... 35

  7.1 Tools ................................................... 35
  7.2 Libraries .............................................. 35
  7.3 Build system .......................................... 35
  7.4 External resources ................................... 35
Analysis and visualization of DNA sequences using cloud computing

8 Framework Implementation

8.1 Package diagram

8.2 Plugin development

8.3 External tools

8.4 Aligner

8.4.1 Indexed aligner

8.4.2 Non-indexed aligner

8.5 Uniqueness map

8.6 Variant detector

8.7 Data encoding / compression

8.7.1 Fasta reference

8.7.2 Aligned Sequence

9 Visualization Implementation

9.1 Resource caching

9.2 Paired-end visualization

9.3 Various improvements

9.4 Variant reporting

9.4.1 Café Rouge

9.4.2 Framework

9.4.3 HTML report

9.5 Annotation display

10 Performance tests

10.1 Aligner

10.1.1 Indexed aligners

10.1.2 Non indexed aligners

10.2 Variant scan

10.3 Uniqueness map

10.4 Conclusion

11 Results

11.0.1 Framework results

11.0.2 Distributed computing results

11.0.3 Visualization results

12 Future improvements

12.1 Faster uniqueness map creation

12.2 Multilayer parallelization

12.3 Align indels

12.4 Command line tools

12.5 User testing

12.6 Better archiving

12.7 Speed improvements

12.8 Various improvements

13 POP-Java conclusion

14 Conclusion

A CD-ROM content
### Data formats

- **B.1** Varscan ........................................... 75
- **B.2** BAM ............................................... 76
- **B.3** Wiggle ............................................. 76
- **B.4** BED ............................................... 76
- **B.5** GFF ............................................... 77
- **B.6** Fasta ............................................. 77
- **B.7** FastQ ............................................. 77

### Manual

- **C.1** Installation & Updates .......................... 79
- **C.2** Framework configuration ........................ 79
- **C.3** Patient management ............................. 80
- **C.4** Data importing .................................. 82
- **C.5** Sequence alignment ............................. 84
- **C.6** Data analysis ................................... 84
- **C.7** Data visualization ............................... 88
  - **C.7.1** Navigation .................................... 88
  - **C.7.2** Information .................................... 89
  - **C.7.3** Genes ........................................ 91
  - **C.7.4** Variations .................................... 92
  - **C.7.5** Report generation ............................ 93

### Planning

- **D** Planning ........................................ 95
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Visualization application after semester project</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>A gene composed by different exons</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>The main page of the galaxy framework</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Activity diagram of the reference scenario</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>Use case diagram</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>Component diagram</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>Deployment diagram</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Distributed infrastructure</td>
<td>27</td>
</tr>
<tr>
<td>9</td>
<td>Basic sequence alignment algorithm</td>
<td>28</td>
</tr>
<tr>
<td>10</td>
<td>Basic sequence alignment algorithm with Pop</td>
<td>29</td>
</tr>
<tr>
<td>11</td>
<td>Hashtable used for the index</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>Indexed alignment algorithm</td>
<td>31</td>
</tr>
<tr>
<td>13</td>
<td>Uniqueness map creation algorithm</td>
<td>32</td>
</tr>
<tr>
<td>14</td>
<td>Uniqueness map creation algorithm with Pop</td>
<td>33</td>
</tr>
<tr>
<td>15</td>
<td>Ensembl web interface</td>
<td>36</td>
</tr>
<tr>
<td>16</td>
<td>Simplified package diagram</td>
<td>37</td>
</tr>
<tr>
<td>17</td>
<td>Plugin class diagram</td>
<td>38</td>
</tr>
<tr>
<td>18</td>
<td>Indexed alignment class diagram</td>
<td>39</td>
</tr>
<tr>
<td>19</td>
<td>Skipping the end of a aligned sequence</td>
<td>41</td>
</tr>
<tr>
<td>20</td>
<td>Non-Indexed alignment class diagram</td>
<td>42</td>
</tr>
<tr>
<td>21</td>
<td>Uniqueness map creation class diagram</td>
<td>44</td>
</tr>
<tr>
<td>22</td>
<td>Cache Sequence diagram</td>
<td>48</td>
</tr>
<tr>
<td>23</td>
<td>Paired ending visualization</td>
<td>49</td>
</tr>
<tr>
<td>24</td>
<td>Uniqueness map of a reference sequence</td>
<td>50</td>
</tr>
<tr>
<td>25</td>
<td>Coverage graph</td>
<td>50</td>
</tr>
<tr>
<td>26</td>
<td>Saved variants displayed in the framework</td>
<td>52</td>
</tr>
<tr>
<td>27</td>
<td>Variant report as a html document</td>
<td>52</td>
</tr>
<tr>
<td>28</td>
<td>Example of a wiggle file</td>
<td>53</td>
</tr>
<tr>
<td>29</td>
<td>Bed annotation file</td>
<td>53</td>
</tr>
<tr>
<td>30</td>
<td>Integer test execution times</td>
<td>55</td>
</tr>
<tr>
<td>31</td>
<td>Indexed alignment speed comparison</td>
<td>56</td>
</tr>
<tr>
<td>32</td>
<td>Indexed alignment speedup</td>
<td>57</td>
</tr>
<tr>
<td>33</td>
<td>Indexed alignment overhead</td>
<td>58</td>
</tr>
<tr>
<td>34</td>
<td>Non indexed aligner speed</td>
<td>59</td>
</tr>
<tr>
<td>35</td>
<td>Aligners scaling over multiple cores/machines</td>
<td>60</td>
</tr>
<tr>
<td>36</td>
<td>Speedup graph for the non indexed aligners</td>
<td>61</td>
</tr>
<tr>
<td>37</td>
<td>Integer test execution times</td>
<td>62</td>
</tr>
<tr>
<td>38</td>
<td>Uniqueness scaling over different threads</td>
<td>63</td>
</tr>
<tr>
<td>39</td>
<td>Speedup values for both uniqueness graph implementations</td>
<td>64</td>
</tr>
<tr>
<td>40</td>
<td>Uniqueness scaling over different threads</td>
<td>65</td>
</tr>
<tr>
<td>41</td>
<td>Speedup values for both uniqueness graph implementations</td>
<td>65</td>
</tr>
<tr>
<td>42</td>
<td>Framework main window</td>
<td>68</td>
</tr>
<tr>
<td>43</td>
<td>Visualization window</td>
<td>69</td>
</tr>
<tr>
<td>44</td>
<td>Configuration dialog</td>
<td>79</td>
</tr>
<tr>
<td>45</td>
<td>Reference management dialog</td>
<td>80</td>
</tr>
<tr>
<td>46</td>
<td>Ensemble gene import</td>
<td>80</td>
</tr>
<tr>
<td>47</td>
<td>Patient explorer</td>
<td>81</td>
</tr>
<tr>
<td>48</td>
<td>Patient explorer popup</td>
<td>81</td>
</tr>
<tr>
<td>49</td>
<td>Raw data importer start</td>
<td>82</td>
</tr>
</tbody>
</table>
Analysis and visualization of DNA sequences using cloud computing

50 Raw data importer quality control
51 Raw data importer preprocess
52 Raw data importer filter
53 Raw data importer saving
54 Alignment import
55 Raw read display
56 Bowtie aligner options
57 Alignment display
58 Varscan options
59 Status bar to navigate
60 Information display
61 Quality viewer
62 Sequence visualization
63 General information
64 Base information
65 Sequence information
66 Genes overview
67 Gene selector
68 Gene selector
69 Variants dialog
70 Variant report dialog
71 Variant report finalization
72 Planning of the master project

82
83
83
84
84
85
85
86
86
88
89
89
90
90
91
91
91
92
92
93
94
95
1 Context

After finishing the semester project “Visualization of DNA sequence coverage” which had as a goal to create a visualization application that integrated different resources to allow the users of the application to analyze genetic sequencing data more easily, the need for an application that not only visualizes the final data, but that guides the user through the whole process of analyzing, from the raw data, to the final analysis report, became apparent. The visualization application will also be enhanced further, to address the needs that arise from the complete work-chain. Figure 1 shows the state the visualization application had after the semester project.

![Visualization application after semester project](image)

Figure 1: Visualization application after semester project

This project was done in collaboration with Phenosystems SA and two European research programs, NMDChip [4] and Gen2Phen [2]. While Phenosystems SA was actively involved in the development process, the two European research programs were consulted for feedback about the actual application and the features needed. Taken from the NMDChip homepage [4], here is its description:

“The NMD-Chip project was funded by the European Commission under its seventh framework programme (FP7) under the call for 'High throughput molecular diagnostics in individual patients for genetic diseases with heterogeneous clinical presentation'.

The aim of NMD-Chip is to design, develop and validate new sensitive high throughput DNA arrays to efficiently diagnose patients affected by neuromuscular disorders. The project is focusing on Duchenne / Becker Muscular Dystrophies (DMD/BMD), Limb Girdle Muscular Dystrophies (LGMD), Congenital Muscular Dystrophies (CMD), and Hereditary Motor-Sensory Neuropathies and Charcot-Marie-Tooth Neuropathies (CMT).”

Gen2Phen has a more data centric approach, and created for example the Café Rouge [6] project. Here is the description of the project taken from their webpage [2]:

The GEN2PHEN project aims to unify human and model organism genetic variation databases towards increasingly holistic views into Genotype-To-Phenotype (G2P) data, and to link this system into other biomedical knowledge sources via genome browser functionality. The project will establish
the technological building-blocks needed for the evolution of today’s diverse G2P databases into a future seamless G2P biomedical knowledge environment, by the projects end. This will consist of a European-centred but globally-networked hierarchy of bioinformatics GRID-linked databases, tools and standards, all tied into the Ensembl genome browser.

Before going into the details of what this project is about, first a small introduction about the world of genetics and the problems present in this domain.

The human genome is composed of 24 different chromosomes. A chromosome can be imagined as a long string of characters, containing only 4 types of characters, or so called bases. Those bases are called A,C,T and G. Each chromosome contains between 46 million and 246 million bases. In the language of computer size this equals text files containing a chromosome that weight between 46 or 246 MB. The order of the bases can be seen as a biological computer program, determining the proteins etc. produced by each cell of the human body. Today, the understanding of those processes is advanced in some parts and limited in others. What we do understand is that there are regions in the genome that work together, we call them genes, to create proteins or other important elements that form our body. A gene is usually composed of different parts, sections in the genome that are called exons, that are used together. Figure 2 shows a gene with its exons. The distance between 2 exons can be anywhere between a few bases or several thousand bases.

![Exons Diagram](Figure 2: A gene composed by different exons)

A gene can have different transcripts. A transcript is a combination of exons that are used together to create for example a protein. When transcribing a transcript into a protein, the space between the exons is ignored and the different exons are put together as one long sequence for the transcription. Not every part of the transcript is used for the protein, there are so called coding and non coding areas of the transcript. The non coding region at the start of a transcript is called UTR 5' and the one at the end UTR 3'.

What a geneticist is usually interested in are mutations, so called variants, in those exons, or close the border of an exon, that could affect the protein that is created by this gene.

After having sufficiently defined what the human genome is in the context of this project, we need to look at how that biological data, composed of atoms, comes from the real to the digital world. The process of digitalizing DNA is called sequencing. Different techniques are available and new ones are developed as we speak. I will only explain the general concept of sequencing, because what is of interest for us is not the exact process of the sequencing, but the final result.

As mentioned earlier, a human chromosome is basically a sequence of a few million bases. Those bases cannot be read all at once, the current technology makes this impossible. What is currently done is to break up the chromosome into small pieces, depending on the technology between a few bases(around 30) or a few hundred bases long each, and then read those little pieces. Because of reading errors, more than one read for every region of the chromosome is done. The result is a list of sequences of different lengths, where the location in the chromosome is unknown, that might overlap.

Now that the biological data has been digitalized, it needs to be aligned. This is the process of identifying the position of each of the sequences that has been read on the original chromosome. In order to accomplish this, a reference genome is used. The reference sequence is the genome of a
theoretical human, created by mixing the genome of several people. To align the sequences, an algorithm basically searches for the most similar region in the genome for each sequence that has been read.

Afterwards the differences of the sequences that have been read and the reference sequence are searched for and analyzed to see which ones affect certain genes of interest.

With the biological background in place, we look more on the technological context of this project. Several applications exist to solve the problems that present on the different steps of the bioinformatics side of the process. They do a reasonably good job at what they do. The major problem with the situation today is the lack of integration between the different tools and the technical knowledge needed to use those tools. Most tools are specialized to one task, for example alignment or variant detection.

One of the problems is that many calculations in bio-informatics are very resource intensive. For example, alignments can take a very long time when a large amount of sequences need to be aligned. This is where more an more popular technologies like grid and cloud computing come into play. The College of Engineering and Architecture of Fribourg under the lead of Prof. Pierre Kuonen created a framework to develop grid computing applications without having to worry about the technical details of how to distribute the load over the network. The POP framework is originally a C++ framework, but has been ported to Java by Valentin Clement, a former student, during his POP-Java Bachelor project. The POP-Java project is not yet mature and needs further testing beyond simple test applications. Through this project parallel algorithms will be developed to test POP-Java and to see what the benefits and problems a parallel approach to the problems in bio-informatics are.
2 State of the art

As stated, different tools exist for different tasks of the analysis process in genetics. There are also tools that regroup them into one framework and abstract the usage to the final user. A few approaches to grid computing can also be found. The following sections will describe what tools and applications currently exists.

2.1 Frameworks

First we look at the tools that implement the whole workflow, similar to our project. There are not many frameworks that have a similar goal to ours.

2.1.1 GAPSS

GAPSS [10] is a pipeline developed by the Leyden university. The goal is similar to our approach. The raw data is taken, aligned and scanned for variants. For the visualization part, the UCSC online browser is used. The UCSC browser is a web portal provided by the University of California Santa Cruz. To visualize the data on the UCSC website, Wiggle files, which are annotation files for genetic data, are created by GAPSS, are created and uploaded to the website. GAPSS has no support to distribute the load of the aligner over several computers, nor does it provide a way to do the alignment simulation.

Website: http://www.lgtc.nl/GAPSS/

2.1.2 Galaxy

Galaxy [11] again has a similar goal than we do, but without the visualization. Galaxy is a web-application, where the user can make different manipulations on his data an create output files depending his needs. The different worksteps can be saved, so that they can be repeated easily. The final result can then be viewed in any dataviewer that supports the data files, but often the UCSC online browser is used. Figure 3 shows the main interface of the application.

Figure 3: The main page of the galaxy framework

Website: http://galaxy.psu.edu/
2.2 Distributed tools

In this section, we look at different projects that tried to use the gridcomputing approach in the domain of bio-informatics. All of those tools try to solve a particular task and do not try to offer a complete solution like the framework we have seen before.

2.2.1 JCVI cloud BIO-Linux

Even though the JCVI cloud BIO-Linux project is not committed to create a entire pipeline, it is still interesting because it brought the alignment to the amazon cloud service. The idea of JCVI cloud BIO-Linux is to create a linux distribution that has all the tools needed to perform different bio-informatics tasks. That distribution is setup in a way that it can run on the Amazon cloud service. While that way it is possible to use the power of the cloud, but no special algorithms are implemented to use the possibilities of a distributed environment to its full extent.

Website: www.jcvi.org/cms/research/projects/jcvi-cloud-biolinux/

2.2.2 Cloudburst

Cloudburst [12] is a parallel sequence alignment algorithm developed by the Maryland University. Like the cloud Bio-Linux, this project does not provide the entire toolchain, but only the alignment. The alignment is done in a parallel way that does scale linearly over the number of processors and reads that need to be aligned.

Cloudburst uses the Hadoop framework developed by Apache [32] to handle the gridcomputing part. The application is written in Java. To use the application, on every computer belonging to the grid the Hadoop framework needs to be installed. Afterwards the client can run in a few easy steps the alignment from his computer.

Website: http://cloudburst-bio.sourceforge.net

2.2.3 Digipede

Digipede [13] is a GridComputing framework not unlike POP-Java, but built on the .Net framework. Digipede allows the distribution of .Net objects over the network and the administration of the grid created this way. While not relevant to our project because of the technology used, it is still interesting because it has been used by the Friedrich Miescher Institute as a part of the Novartis Research Fundation, in the domain of bio-informatics. They used Digipede to improve the performance of a webservice they offered through their website. The ability to perform the heavy computations on a distributed network improved not only the performance of the webservice, but also the performance of the website that was hosted on the same webservice, because the server was not overloaded anymore by the calculations needed for the webservice.

Website: http://www.digipede.net

2.3 Specialized tools

In this section different we describe specialized tools that usually perform one specific task. Some of those tools will be reused in this project. Those tools are not distributed, but might use multiple cores of a single machine.
2.3.1 Samtools

The samtools [19] are a collection of useful utilities. They are used to interact with the SAM/BAM fileformat [B.2]. The fileformat used in this project for alignment data is the BAM fileformat. But many tools use the SAM fileformat as their output format. The samtools have the following tools integrated:

- Convert SAM file to BAM files and back.
- Sort SAM and BAM files.
- Index a sorted SAM or BAM file.
- Merge SAM or BAM files.
- Create a special pileup file used for variant detection.
- Simple text bases alignment visualization.

The conversion, sorting and indexing tools are used by the framework as is the pileup file creation.

Website: http://samtools.sourceforge.net

2.3.2 Varscan

Varscan [20] is a Java application used in collaboration with samtools [19] to detect the variants in a BAM file. It is able to detect all kinds of variants, most importantly insertions and deletions, so called indels. To do this, it uses the pileup file created by samtools. Varscan allows to scan for variant using various options. The most important option is to filter the variants with a minimum coverage, a minimum frequency and also based on the quality of the bases in the alignments. This allows to filter the variants down to a reasonable number to check manually.

Website: http://varscan.sourceforge.net

2.3.3 Bowtie

Bowtie [21] is a widely used aligner. It is actively developed and known for its speed. It is based on a Burrows-Wheeler index [22] which allows a low memory footprint. One of the downsides of bowtie is that it is not very good at aligning sequences with insertions or deletions. This can be a real problem, because certain sequencers have reading errors that create insertions or deletions. Bowtie is not able to map those reads and loses information because of this.

Website: http://bowtie-bio.sourceforge.net

2.3.4 BWA

BWA [23] is another aligner. Similar to bowtie it is very fast, and also based on a Burrows-Wheeler index [22] and is actively developed. In contrast to bowtie the alignment quality of sequences with insertions and deletions is very good, which makes it more interesting to use on certain types of sequencers.

Website: http://bio-bwa.sourceforge.net
2.3.5 Stampy

Stampy [24] is a relatively new aligner with good results for sequences with indels. While it is reasonably fast, it is not as fast as BWA or Bowtie. It does have the option to be used with BWA to speed up the processing.

Website: http://www.well.ox.ac.uk/project-stampy
3 POP-Java

POP-Java as already mentioned is an in-house development at the College of Engineering and Architecture of Fribourg lead by Pierre Kuonen. It allows the programmer to develop gridcomputing applications without having to worry about the grid itself. The pop environment started with a C++ implementation, POP-C++ [15], that has been freshly joined by a Java implementation. The two implementation are compatible, which means that a POP-Java application can use a POP-C++ object and the other way around. The main advantages of the Pop environment are the ease in which parallel objects can be created, and the way they are distributed over the network.

The main characteristic of POP-Java on the code side is how the Java code is made into a parallel object. A POP-Java class is not defined with the normal class keyword, but instead with the parclass keyword. Furthermore, every public function uses two additional keywords in addition to the two standard Java ones (visibility and return type), the concurrency keyword and the synchronicity keyword. Every function in POP-Java can either be synchronous or asynchronous. This means, that synchronous functions behave just like normal Java functions, where the caller waits for the return value before continuing its code. On the other hand, asynchronous functions will return immediately, letting the original caller continue its code. Of course this does not allow any return value in the asynchronous functions.

The other keyword, the concurrency keyword, defines the calling order of the functions. In every parallel environment, function calls can come at every moment. Instead of using semaphores as they are usually used when implementing parallel objects with threads, POP-Java allows the programmer to define if a function is concurrent, sequential or a mutex function. Concurrent functions can be called all the time, as long as no mutex function is currently executing. Sequential functions can be executed as long as no other sequential or mutex functions are executing. Mutex functions can only be executed if no other function is currently executing.

This ability to define how the functions work in a parallel environment makes the code easier to understand, and is thanks to this less error prone and removes the need for most concurrency control mechanism, like semaphores. The resulting code looks like the following example:

```java
public parclass Integer {
    private int value;
    public Integer() {
        value = 0;
    }

    public sync conc int get() {
        return value;
    }

    public sync mutex void add(Integer i) {
        value += i.get();
    }

    public async seq void set(int val) {
        value = val;
    }
}
```
This example, Integer class, can be used in a grid environment. The class uses all the concepts just presented. As can be seen, the code is much easier to read as a similar code in pure Java would be. The code of this example has been taken from the POP-Java manual written by Valentin Clément.

The other main feature of POP-Java is how the objects are distributed over the network. The programmer does not have to worry about how the distribution is made, and how the communication between the objects is done. A new object can be created like every other Java object. At the creation of the object with “new”, the pop environment will find a suitable machine in the cluster and transfer the object there. POP-Java allows the programmer to define some criteria, like processing power or available memory, that are required on the machine on which the object will be created.
4 Specification

The basic environment in which this project is created has been described from a technological and a practical point of view. In this chapter, we define the actual project, first by a description of what it should do and then more specifically with the individual goals we want to accomplish.

The main goal of this project is to create a framework that allows a non technical person to perform all the steps needed for the analysis of a genome from the raw data coming out of the aligner to the final report that can be saved. Those steps include filtering the raw data, aligning it, scanning for variants, performing further analysis on the data, visualizing it and creating a report. For those tasks, external applications will be used wherever possible, but an alternative will be implemented in case no external application is available. The user interface should be comprehensible to a beginner, but also allow an advanced user the flexibility he needs. During the implementation, the possibilities of distributed computing, in particular POP-Java, will be explored and compared to a traditional threaded implementation.

4.1 Goals

After defining all tasks and having set the general context, we need to define the actual goals of the project. Those goals can be reduced to a few groups: the creation of a framework to support the workflow, the integration of distributed computing into the framework and the improvement and integration of the already existing visualization tool. Those are the big goals. Other tasks like Variant detection, quality assurance and other steps need to be done too, but they are more subtasks of the other tasks.

4.1.1 Framework

The idea of the framework is to create an application that can run the different steps of the workflow, either by using data directly provided by the user, or by channeling data from one working step to the next. For the different steps, custom developed software or external applications can be used. The user should be able to manage the different patients he has with their different reads and alignments. The real goal here is to be better and more specialized than our competition. Not many complete frameworks exist, and the most used one, Galaxy [11], is very complicated and not user friendly at all. We do not want to integrate as many tools as Galaxy does, simply because this would by far exceed the possibilities of this project, but we want to integrate them in a manner that is easier to use and better integrated. An additional goal of this framework is to increase the value of the visualization application developed in the previous semester project.

4.1.2 Distributed computing integration

As we could see, the main goal of the distributed computing integration will be to implement the alignment algorithm. Not only the implementation is important, but also the business model around it. Several situations can be imagined where the ability to run a computer resource hungry calculation on many computers. For example, small laboratories might not have the money to align the data for themselves because the costs to own a powerful enough computer might just be too big.

An additional goal of the distributed computing integration is to test POP-Java and work closely with its creator to make the needed improvements. Because POP-Java has only been tested so far with simple test applications, this will be the first real life test of POP-Java, and while many problems might arise that could hinder the usefulness of POP-Java during this project, it most certainly will allow POP-Java to progress and become a viable product.
Compared to our competition, we will not try to create a competitive implementation of certain algorithms, but focus more on the feasibility and the specific usefulness of POP-Java in this context.

4.1.3 Visualization tool

The visualization tool created during the semester project is a good base for the task. The limited time during the semester project only allowed to create a good foundation, but for the tool to be usable in a real environment, several improvements need to be made.

During the visits of the Gen2Phen [2] and the NMD-Chip [4] project meetings, which are both european research projects, many feedbacks could be gathered by potential endusers of the application. The following important features were identified:

- Creation of a Variant report
- Ability to submit a new variant to a central database like Café Rouge [6]
- Better performance for all features of currently supported fileformats (BAM, varsan, wiggle, bed)
- Caching of external resources to work offline
- Speed improvements for slow internet connections
- Paired-end read support
- Various usability improvements
- Bugfixes and stabilization

4.1.4 Usability

One of the goals is to ensure that the application is user friendly and not confusing for non specialized people. While a biological background is required to use the application, a computer science background is not.
5 Analysis

Before starting the implementation, we have to understand how the whole process and each of its steps work.

5.1 Reference scenario

The basic idea of a genetic diagnosis is that a patient goes to his doctor, which then makes a genetic analysis and gives the patient a diagnosis. We are now going to break this process further down into its components.

First, the patient consults the doctor. Usually a tissue or blood sample will be taken and then be sent to a laboratory. The laboratory will bring the sample into a form that can be sequenced. This can be a simple process, but could also involve the mixing of different patient samples to sequence them at the same time.

After the biological treatment of the sample has been done, it is sequenced. The sequencing can be done internally or externally, but this is not important for our purpose. During sequencing, the DNA contained in the sample will be read in small parts. Each part can be between 20 and 400 bases long, depending on the technology used. This data will then be saved in a raw data format, usually simple text files, and then be sent back to the laboratory. After receiving the sequenced data, the data will be checked for its quality, and then separated if needed into the different patients that were sequenced at the same time. Once the patient data has been separated, it can be analyzed. To do this, the data will first be aligned to the reference sequence. This might be the whole human genome, one chromosome, or only the region surrounding one gene. Afterwards a simulation of the alignment will be done to determine the coverage on an optimal patient.

Once this has been done, the sequenced and aligned data will be searched for variations regarding the reference sequence in an automatic way. Once all this has been done, the biologist can visualize the data and decide what variations are real variations and not artifacts from the sequencing technology. During this step he can consult external resources or launch extensive analyses. With those findings he will then create a report. This report can be archived and sent to the doctor, which will do the final diagnosis.
Figure 4 shows this reference scenario. Some steps were marked as optional, even if they were not described that way in the reference scenario description. The parts marked in green are the ones that this project tries to address. To further understand the process and in what way it can be simplified and improved we need to understand more about the individual steps that are concerned by an automated approach.
5.2 Parallelization

Now that the tasks are defined, we try to identify the steps that would benefit from parallelization, since one of the goals is to test and use POP-Java. The most prominent step that can be run with a parallel algorithm is the alignment algorithm. Due to the nature of the data, it can be run in parallel very effectively. Depending on the dataset size, the alignment can take hours to complete. Using a distributed approach we can hope to reduce the time needed to perform this task. The alignment simulation task is very similar to the normal alignment, which makes that it can be distributed similarly well and can take advantage of the work done to distribute the alignment task.

The data quality check and splitting step are not well suited for parallelization, because they are very fast, and the time that would be needed to send the data into the cloud would be inefficiently bigger than the local treatment. The same goes for the Variation detection step.

The next step that can be parallelized are the calculations that can be run from the Visualization step. Those calculations are mainly decision helping tools, for example to determine the effect of a base change in a gene on the structure of the protein that gene creates.

This leaves us with 3 steps that can be run in parallel:

- Alignment
- Alignment simulation
- Decision helping, for example protein folding
5.3 Use cases

We can identify the following use cases:

![Use case diagram](https://via.placeholder.com/150)

**Figure 5: Use case diagram**

5.3.1 Actors

The following actors are involved in the use case diagram:

- **Laboratory**: The user of the application.
- **Raw data**: The data that comes directly out of the sequencer.
- **Preprocessed data**: The data that was stripped of marked sequences.
- **Reference sequence**: DNA reference against which we align the data.
- **Aligned data**: The raw data aligned to the reference.
- **Variation report**: A report describing all features of a certain variant.
- **External resources**: Webservices like ensemble [8] that provide additional information.
- **Café Rouge**: Central database to save and submit variants.
5.3.2 Use case descriptions

In the following sections the different uses cases are described.

Quality check
Actors: Laboratory, Raw data
Description: During this step, basic checks for the data quality are performed. For example the number of reads and their size (minimal read length, maximal, mean and average) are controlled. The main idea is to give the person using the application a simple way to determine the quality of the data.

Patient data split
Actors: Laboratory, Raw data, Preprocessed data
Description: This step is optional and only used if different patients were sequenced at the same time or if for any other reason the sequences that were read have been marked by a marker DNA. This is not a complicated process. The basic idea is to take the sequence raw data, and split it into different files depending on the tag each sequence carries. That tag will be removed during the process.

Alignment
Actors: Laboratory, Preprocessed data, Reference sequence, Aligned data
Description: After the data has been separated into each patient, the small sequences have now to be mapped to the reference sequence. The human genome has a length of about 3.2 Gbp (Giga base pairs) and one chromosome has on average a length of 140 Mbp (Mega base pairs). To find the correct location of a 20 or 400 bp sequence, is not always trivial. Different algorithms exist to solve this problem, what exact algorithm is best suited will be further explored in a later section of this document 6.3.

Simulation
Actors: Laboratory, Reference sequence
Description: The alignment simulation creates a uniqueness map of the DNA we sequenced. The idea is to find out how unique each sequence is on the reference. The reason this information is important is that if a certain sequence maps to different regions in the reference sequence, then the result of the alignment is unclear. For example, the coverage of a certain base in the reference might be low with the data aligned. The technique used to create this map is very simple. The reference sequence is split up into parts that are of the same size as the example data and those parts are then mapped back to the reference sequence. That way we can find out at how many places are similar to that particular piece.

Variation detection
Actors: Laboratory, Reference sequence, Aligned data
Description: Now that the data has been mapped to the reference sequence, differences can be searched for. During this search, the differences found between the mapped sequenced and the reference, so called variants, are recorded. Different types of variants exist. First there are simple base changes, they are easy to detect. Then single inserts or deletes of bases can be found. Lastly whole portions of the DNA can be deleted or duplicated. Those can potentially be found using the coverage and the uniqueness map generated in the alignment simulation.

Visualization
Actors: Laboratory, Reference sequence, Aligned data
Description: During the visualization, the biologist will check the variants found in the previous check to see if they are real. He will look at the actual data that was sequenced and use all the information he has to determine if this variant is interesting or not.
Report generation

*Actors:* Laboratory, Variation report

*Description:* When a variation has been identified as legitimate, a report is created. The relevant information about the variant such as the coverage, the amount of reads for each base, known variants in public databases at this position etc., are put into a final report that can be sent to the doctor.

Archiving

*Actors:* Laboratory, Café Rouge

*Description:* The report generated in the usecase “Report generation” can be archived. If the actual report in PDF form or only the underlying data is archived, has yet to be determined. Part of the archiving will be the submission of the data to Café Rouge [6], a shared database for mutation information.
6 Design

This chapter defines the framework of how the project will be implemented. The basic approach that is taken for the design of the application is that a central application that coordinates the different steps of the workflow needs to be created. This application needs to provide a graphical user interface to the user and new workflow steps should be able to be added easily and be interchangeable. With this basic concept in mind, we can identify the following components:

![Component Diagram]

Figure 6: Component diagram

As we can see, the framework that needs to be created is in the middle and coordinates the different components which represent the different use cases of the application. The different components connected to the framework need to be interchangeable, to allow for example different aligners to be used. The actual GUI should not care about the underlying components. Because of this, all components are defined as interfaces that hide the underlying implementation. This is not represented directly in the component diagram.
6.1 Deployment

Based on the analysis so far we get the following deployment diagram:

![Deployment Diagram](image)

Figure 7: Deployment diagram

There are 3 main groups in the deployment:

- The Framework and the machine it is working on
- The visualization part of the application
- The distributed environment where some tools of the framework are executed

While the visualization part and the framework usually run on the same computer, it is not necessary because the independence of the visualization application will be preserved. The communication between the framework and the visualization tool will be very one sided. The framework starts the visualization using a configuration file that defines the data that needs to be visualized.

The distributed group contains the components that are executed on a distributed environment. Those components are not necessarily executed in a grid environment, but non-distributed alternatives will be developed.

6.2 Gridcomputing business model

As previously discussed, the integration of gridcomputing in our application is not only an advantage in technical terms to speed up the calculations, but also adds new business opportunities. Many bioinformatics related calculations are very resource intensive and take a long time. Small and even middle sized laboratories do not always have enough computer power at their disposal to execute those calculations in a reasonable timeframe. This is where a new business model can be created. An enterprise like Phenosystems SA, but also universities that have processing power at their disposal, can rent computer time to the laboratories that need it. This benefits the laboratories, they do not need to invest and maintain server farms that will not be used at 100% anyway, and the company renting the processing power is able to use their computers in an efficient and cost effective way. The company renting the processing power would have a public server that can accept the queries, those queries would then be dispatched to different servers, figure 8.
An interesting business model could be to offer the user the choice of how much time he wants to spend on the calculation, and pay depending on his choice. For example, the user might get the choice of doing the calculation in 5 minutes, 30 minutes or 1 hour. Depending on the option he chooses, he will be charged differently. The time proposed could be dynamic, depending on the current load of the grid. While the estimation of the needed time will always be an approximation, and perhaps instead of a actual time the user could have the option between fast, normal and slow, this idea can be interesting for both the user and the company owning the cluster.

One aspect of gridcomputing is security. Especially in the domain of genetics, the data that is worked with is often sensible and private data. Sending it over the network can be a security problem, especially if not all parts of the network are controlled, as it is the case with the internet. This means that the data needs to be encrypted when it is sent over the network. Luckily POP-Java does exactly that, all the data sent over the network is sent over a secure SSH connection.

The remaining problem is the data that is processed on the grid itself. On the machines that compute the data, it is not encrypted. This is a potential security problem for the user of the application, as he has to trust the operator of the grid. Nothing prevents the operator to duplicate the results and save them for himself. Different approaches are being researched today on how to compute sensible data on a grid, but exploring them is out of the scope of this project.

It is important to note that while this analysis of the business model and the security problems involved are only a quick look at the subject, and could be approached in a more detailed manner in a different project.
6.3 Algorithms

In this section, the different algorithms that can be used to solve the problems we face in this project will be described. The main algorithms used are the alignment and the uniqueness map generation algorithm. Both algorithms have been implemented with different techniques, using normal Java with threads, POP-Java, and in the case of the uniqueness map creation, also a mix between POP-Java and POP-C++.

6.3.1 Non-indexed alignment

Aligning sequences to a reference is one of the big tasks in bio-informatics. As a first approach, a simple algorithm is designed. The idea is to align every sequence to every position in the reference sequence. Because we do not know if a sequence was originally read in forward or backward direction, we have to test it twice, once in the form it was read and once with its complement.

To create the complement of a DNA sequence, every base needs to be replaced with its complement, and the sequence needs to be reversed. The complements are A,T and G,C. The complexity of this algorithm is high, the amount of calculations that need to be done are \( n \times l \times 2 \) where \( n \) is the number of sequences to align and \( l \) the length of the reference and the multiplier by 2 is because every sequence needs to be aligned forwards and backwards. This brings us to a complexity of \( O(n \times l) \). We see that the amount of sequences that need to be aligned and the size of the reference play both a big role. With reference lengths up to 250 million bases, and millions of sequences to be aligned, we will quickly attend the limits of the current processing power.

The sequence diagram in figure 9 shows the way the basic alignment algorithm works.

![Sequence Diagram](Visual Paradigm for UML Standard Edition(Ecole d'Ingenieurs et d'architectes de Fribourg)]

Figure 9: Basic sequence alignment algorithm
As we can see, first the algorithm loads the data that is needed, the reference sequence and the sequence to be aligned. Then the jobs are created. Each job receives the data needed, this means the reference and the sequences to align. After every sequence that has been found, the job signals this to the controller, which then loads the sequence depending on its ID from the original sequence file, and saves it with the position found by the aligning job into the BamFile.

The algorithm has been designed to work with a traditional Java implementation that uses threads, and a POP-Java implementation with workers. Both use the same sequence diagram 9. The design works very well with the threaded version, while the POP-Java version works well too, it is not the most optimal solution for the problem, depending on the data used. In the Java implementation, the reference is loaded once into memory and then shared between all threads. In the POP-Java implementation, due to its nature, every job receives its own copy of the reference. The result is that if there are very few sequences to align on a big reference, it would be faster to split the reference and let each job align all sequences to his part of the reference. On the other hand if there are many sequences with a smaller reference sequence, the current approach is faster.

Due to the high dependency of the algorithm on the length of the reference, it is not practical to use it on a long reference. Because of this, the current approach works best in this situation. To better explain how the POP-Java technology was used, we can see in figure 10 a modified sequence diagram.

![Figure 10: Basic sequence alignment algorithm with Pop](image)

We can see the two regions of the algorithm, the local and the remote part. The objects implemented in POP-Java are marked in green. As we can see, only one object with no other dependency than the controller object is executed remotely. For every sequence that is successfully aligned, there is one call from the worker object to the controller object.
As a final note of this alignment algorithm it is to note, that while the usage in pure Java works well, a better solution for POP-Java could be created. To do this, a two layer distribution would be implemented. POP-Java would distribute the different workpackages over the network, but on every machine, normal Java threads would be created that would distribute the load over the different processors/cores of the machine. To be able to compare the POP-Java and Java implementation, the approach described before has been chosen.

### 6.3.2 Indexed alignment

Aligning sequences to a reference sequence can be done in several ways. A popular technique is to create a index of the reference to identify interesting regions more quickly. That technique has been used in this project, additionally to the naive alignment that works without a index. The technique used is the following:

For the reference sequence against which the alignment will be done, a index file is created. To create this index, the reference sequence is partitioned into segments of 12 bases, those 12 bases are then hashed, and a hashtable with the position where this hash was found is created. The number 12 is not a random number, but research has shown that this is the most optimal number for genetic data \(^{28}\). The result is a hashtable where it is easy to find the locations where a certain sequence can be found.

This index can then be used by cutting the sequence that needs to be aligned into pieces of the same length as the pieces in the index, then search the index for the hash of those pieces and align the sequence to the location where the same hash has been found. This increases the speed of the alignment very much, at the cost of a index file.

The basic data structure of the index can be seen in figure 11.

![Hashtable used for the index](image)

**Figure 11: Hashtable used for the index**

The index is a sorted hashtable, where the hash is the hash coming from a certain DNA sequence. With the hash we save all positions where the hash was found in the reference sequence. A single hash value can have multiple locations saved with it. The implementation is flexible towards the hashing algorithm used. Currently the standard Java hashing algorithm for strings is used, but if needed this can be changed easily.

The index is then used for every sequence we want to place to identify where the potential locations in the reference are. At the start of the sequence, multiple pieces are taken, hashed and
searched for in the index. The search is done with a standard binary search algorithm, which means that it is done in $O(\log(l/12))$, where $l$ is the length of the reference sequence, which then is divided by 12, because only multiples of 12 words are . This really brings down the complexity of the algorithm, because the actual alignment of a sequence, once a potential location is known, is not very costly. The overall complexity is $O(n \cdot \log(l/12))$, where $n$ is the number of sequences to align and $l$ the length of the reference sequence. This is much lower than the complexity for the basic alignment algorithm.

Because of time constraints, only a threaded version of the algorithm has been designed. The interaction between the different objects can be found in figure 12. The design of the indexed algorithm is slightly different to the basic aligner. The indexed algorithm does not send all the sequences to align directly to the aligner job, but does send a new sequence to align every time the previous one was aligned. Using a local threaded implementation this does not have a negative effect because of the increased amount of communication between the controller and the aligner job. The advantages are the lower memory requirement, not all sequences to align need to be stored in memory at the same time, and the other advantage is that the sequence that was aligned does not need to be reloaded.
6.3.3 Uniqueness map

The uniqueness map shows the user what parts of a reference genome are unique and what parts are not. As described, the importance of this is for the user to judge whether a region that has less alignments than expected has some kind of a gene defect, or if it is simply by the nature of the alignment that this region has less coverage because its low uniqueness.

The ideal approach to this problem would be to cut the reference in pieces, ideally the medium length of the original sequences that were aligned, and align all those pieces with the same aligned and the same options back to the reference.

This approach was not followed during this project for a few reasons. First, it would mean to create a standard FastQ file that contains all pieces of the original reference. If for example the pieces were 30 bases long each, the FastQ file would be at least 60 times larger, because not only does it contain the sequence we want to align, but also the quality of the sequence, that while not really needed in this usage, is required for the data format. A 100MB reference, which would be a normal chromosome size, the resulting file would be 6GB big. The other option to call the aligner for each sequence, removing the need to create a huge temporary file would actually be worse, because the overhead of loading the index associated with the reference would be present for each sequence we want to align.

The solution for this project was to create a custom algorithm, that while not behaving exactly the same as the original aligner, the potential speed is much better and the differences not that important. The algorithm only needs to load the reference sequence into its memory. Using 3 pointers that point into that same reference, we can calculate the uniqueness map. The basic idea of the algorithm is the following. Starting at a certain position, every position after that position is checked if it is a duplicate of the current region. As soon as a position has been found, we save for the current region and the newly found region that there is an additional duplicate. That way, we use the memory in a very efficient way. The algorithm has a complexity of $O(n^2)$, where $n$ is the length of the reference sequence. The sequence diagram 13 shows the communication between the different objects involved in the algorithm.

![Sequence diagram](image)

Figure 13: Uniqueness map creation algorithm

The design has been made for a traditional Java implementation using threads and a POP-Java implementation using workers written in POP-Java or POP-C++. For all implementations the same algorithm has been used, to be able to compare them better, even if a better approach for POP would
exist 6.3.1. All three implementations have use the same sequence diagram as shown in figure 13. To better understand which parts of the algorithm use the Pop technology, we look at the sequence diagram in figure 14.

![Sequence Diagram](image)

Figure 14: Uniqueness map creation algorithm with pop

We can see two main blocks of the algorithm, the locally and the remotely executed one. The objects written with the pop technology are marked in green, while the traditional Java objects are blue. As we can see, the remote object is written with a Pop technology, either POP-Java or POP-C++. The local controller is written in POP-Java. The remote object has as its only dependency the controller object. This way it is simpler to distribute it over the network. The whole calculation is done in the remote object. This is why in the implementation where both POP-Java and POP-C++ were used, the remote object was implemented in POP-C++, because in general, C++ code should be faster than Java code.
One of the bigger problems of this algorithm is the fair distribution of the work that each processor core has to do. The alignment algorithm just had to distribute the amount of sequences to align evenly over each core. This is slightly more difficult with the uniqueness algorithm. The total work that needs to be done by all workers can be explained the following way. Every position of the reference sequence needs to be compared with every other position of the reference sequence that comes after itself. The amount of work can be calculated with the following formula, where $L$ is the total length of the reference sequence:

$$TotalWork = \frac{L^2}{2}$$

From this the amount of work each worker needs to do can be easily calculated as, where $t$ is the amount of workers used:

$$w = \frac{TotalWork}{t}$$

The actual work done by each worker can be calculated by the size of the reference he has to cover and the length of the reference sequence from the start of this region until its end. This leads us to the following formula, where $L_u$ is the length of the reference sequence after the section attributed to the particular worker, and $x$ the length of that region.

$$w = (L_u + (x/2))x$$

Using the last two equations, we can calculate the length of each region that is attributed to each worker, simply by resolving the equation, which gives us this formula as a result:

$$x = \sqrt[2]{2w + L_u} - L_u$$

The algorithm will assign the different regions to the workers starting from the end of the reference sequence. This means that the first region will have a $L_u$ of 0, very much simplifying the first step of the calculation to the following formula:

$$x = \sqrt{2w}$$


7 Implementation

This section of the report will explain how the individual parts of the project have been implemented.

7.1 Tools

The tools needed for the development where similar to the ones used during the first project [31]. Again Eclipse [16] was used as the IDE. Again several plugins where used inside of eclipse, where JUnit [17] and the JUnit [18] benchmark tool were the most useful. As the main development OS, linux has been used because of personal preferences, but also because most bioinformatics related applications only run under linux. As the versioning system, Git [29] has been used. The eclipse plugin EGit [30] was used to integrate the versioning system into the development environment.

7.2 Libraries

The most important library used in this project is the Picard library [26], allows to access BAM files through Java.

7.3 Build system

As the build system, and has been used. Ant is the equivalent of makefiles for C but written in Java and used mainly in Java applications. Using ant allowed the automation of many processes. A prime example is the creation of a release that can be sent directly to the end user. With ant all the steps needed to create the release have been automated.

- Compilation of the pure Java code.
- Compilation of the POP-Java code.
- Launching of unit tests to ensure that the basic features work.
- Creation of a releasable jar file that includes the icons needed by the application.
- Signing of the jar to be usable as a Webstart application.
- Creation of a release folder containing all needed files.
- Creating a tar.gz archive of that folder.

Other processes that are automated are for example the launching of the framework as a POP-Java application. Using Ant increases the development speed very much, having not to redo all those steps by hand is a very efficient way to save time.

7.4 External resources

The viewer application already had a integration with Ensembl [8] to receive meta information about different genes and variations. This integration has been further expanded with the framework, allowing the user to directly download the reference sequence of a gene into the framework. This allows the user to easily manage different genes in the application, without the need to search for the reference on the internet by himself.

Ensembl is a website that contains various information about the human genome, such as the different genes and variants that are currently known. To access the information, there is a web-based interface to access the information that can be seen in figure [15]
To access the information from an application, the ensembl website offers a simple webpage that takes a XML file as a parameter and returns the result of the query. An example xml file that queries all the genes on chromosome 21 can be seen in the following example:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<DOCTYPE Query>
<Query virtualSchemaName = "default" formatter = "TSV" header = "0"
uniqueRows = "0" count = "" datasetConfigVersion = "0.6" >
  <Dataset name = "hsapiens_gene_ensembl" interface = "default" >
    <Filter name = "chromosome_name" value = "21" />
    <Attribute name = "ensmbll_gene_id" />
    <Attribute name = "start_position" />
    <Attribute name = "end_position" />
    <Attribute name = "external_gene_id" />
  </Dataset>
</Query>
```

The result is a list containing the requested attributes. Here are the first lines that the example from before generates:

<table>
<thead>
<tr>
<th>Ensembl Gene ID</th>
<th>Gene Start (bp)</th>
<th>Gene End (bp)</th>
<th>Associated Gene Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENSG00000244123</td>
<td>37263094</td>
<td>37263155</td>
<td>AF020802.3</td>
</tr>
<tr>
<td>ENSG00000243001</td>
<td>37263159</td>
<td>37263224</td>
<td>AF020802.2</td>
</tr>
<tr>
<td>ENSG00000239259</td>
<td>37263329</td>
<td>37263398</td>
<td>AF020802.4</td>
</tr>
<tr>
<td>ENSG00000243611</td>
<td>37263405</td>
<td>37263470</td>
<td>AF020802.1</td>
</tr>
<tr>
<td>ENSG0000024357</td>
<td>45894981</td>
<td>45895050</td>
<td>AP001065.1</td>
</tr>
<tr>
<td>ENSG00000201812</td>
<td>15443192</td>
<td>15443307</td>
<td>5S_rRNA</td>
</tr>
</tbody>
</table>

Figure 15: Ensembl web interface
8 Framework Implementation

The framework is a big application. It consists of a GUI that allows the user to interact with the application, backends to several internal or external tools and a system to manage the data (like patients, alignments etc) that is created during its use. This chapter will explain how the framework was implemented, from the integration of external application to the creation of the custom algorithms described in the Algorithms chapter 6.3.

8.1 Package diagram

To better understand how the framework was implemented there is a simplified package diagram in figure 16.

![Simplified package diagram](image)

Figure 16: Simplified package diagram

This package diagram does not contain all packages present in the application, but only the most important ones that help the understanding of the code structure. With a total of 95 packages, drawing a complete representation of the code would have been more confusing than helpful. What we can see from this diagram is that there are two main blocks, the framework package and the visualization package. The framework package uses the visualization package, to visualize the data, while the visualization package works in an independent way from the framework. Both packages share the lib and tools package, which contain code that is shared between the two. The lib package contains more data and gui related code that is shared, for example a class that accesses a fasta file, which is used by both the framework and the visualization package, or common widgets, for example to select a human chromosome. The tools package on the other hand has little utilities, like a way to log information or having different types of console output like debug and warning, where the debug output can be disabled application wide using one variable.
8.2 Plugin development

The framework is an interface to the data we want to analyze and the tools we want to use to do this. Those tools might run several minutes or hours. During the development, several new tools might be added and it is not really possible to foresee what kind of tools will be added in the future. Because of this reason, an abstraction for those them was created.

Every tool that is used, an custom coded or a external one, must implement the FrameworkPlugin interface. The FrameworkPlugin itself implements the Java Runnable interface, that way every plugin can be started as a thread, not interfering with the main thread running the GUI. The structure of the plugin infrastructure can be seen in figure 17.

![Figure 17: Plugin class diagram](image)

Several other interfaces exist that the plugins can implement, one for the aligners, one for the coverage scanners, and others for the different functionalities the framework offers. Thanks to this, it is trivial to add a new plugin, for example to do the alignment and integrate it into the framework. The GUI is dynamic and will use the plugins provided by the FrameworkControll object.

8.3 External tools

Several bioinformatics tools have been integrated into the framework. Those tools are used for the sequence alignment and the variant detection. The actual tool description can be found in the section state of the art 2.3. The following external tools were finally used:

- Samtools, as a general purpose tool handling SAM files.
- Varscan, as a variant detector.
- Bowtie, as an aligner.
- BWA, as an aligner.
- Stampy, as an aligner.
8.4 Aligner

Two different aligners have been developed, as described in the algorithm chapter 6.3 of this report, a indexed and a non indexed aligner. The non indexed aligner was implemented two times, once in normal Java and once with the POP-Java technology to distribute the calculations over multiple computers. The aligners use a Fasta 3.6 file as the input for the reference sequence.

8.4.1 Indexed aligner

The indexed aligner makes use of a index of the reference sequence to speed up the alignment. This is done using a hashtable that allows the quick identification of possible positions where the sequence could be located, the actual alignment is then done only on these positions. The details of this algorithm can be found in the algorithm chapter 6.3.2. The implementation of the algorithm uses 3 classes, as described in the design phase. Figure 18 shows a highly simplified version of the class diagram, where only the classes and functions relevant to the implementation of the algorithm are described. Classes used for the framework integration, or the loading and saving of the data files are not shown in this diagram.

Figure 18: Indexed alignment class diagram

The main class which takes the role of the controller is the CustomIndexedAligner class. This class creates a index object which is then used by the CustomIndexedAlignerJob to perform the alignment. The CustomIndexedAlignerJob will use the getIndexes function of the CustomIndex to get a list of all possible positions of a certain sequence. As soon as the position was found, the foundAlignment function of the CustomIndexedAligner is called to store this alignment. Before running an alignment, the controller class first sets the sequence to be aligned on the aligner job with setSequence.

The most interesting part of the code is probably how the index of the reference sequence is created. The following code sequence shows how the list of all hashes and their position is created.

```java
//Load reference file
FastaSequential seq = new FastaSequential(fasta);
```
int total = length / PIECE_LENGTH;

//temporary list with all hashes and their position
Pair<Integer, Integer>[] tempArray =
    new Pair[total];
String piece;
int index = 0;
int hash;
try{
    //create hash for every subsequence of the reference
    int i = 0;
    for (; i <= length - PIECE_LENGTH; i+= PIECE_LENGTH){
        piece = seq.getNextBases(PIECE_LENGTH);
        hash = piece.hashCode();
        tempArray[index] = new Pair<Integer, Integer>(hash, i);
        index++;
    }
    seq.close();
} catch (IOException e){
    return false;
}

Arrays.sort(tempArray);

saveToFile(indexFile, tempArray);

The code is simple, yet effective. Much care has been taken to make this code as efficient as possible, from the point of view of processing power and also memory consumption. The one thing that we can see in that code that causes the highest amount of memory consumption is the Pair object created for every element. Sadly there is no way around this, because of the need to sort the hashes at the end, and the positions they are associated with need to be identifiable after the sorting. The code will create the list of all hashes and their position, this means that if a hash is present multiple times, it will appear multiple times in that list. Because of this, when saving the list to the actual file, all duplicates are merged. To achieve this, the list is sorted, so that all entries with the same hash are grouped together.
One of the additional features this aligner has that the non indexed aligner does not have, is the ability to mark the end of a sequence as skipped, if up to there the amount of errors was under a certain level. The user can choose, for example, to say that after 75% of a sequence, if the errors found where under the maximum amount specified, then the sequence is considered as aligned and if the error count goes over the maximum allowed after this 75% of the sequence, the rest of the sequence is marked as skipped. The advantage of this is that many sequencers produce more and more errors towards the end of a sequence, and with this technique, the rest of the sequence can still be aligned, while the variant detector will not report false variants. The visual result of this can be seen in the viewer in figure 19.

Figure 19: Skipping the end of a aligned sequence

To implement this, the aligner used the property of the BAM alignment file called CIGAR string. The CIGAR string allows the aligner to define which bases of the sequences were a match, a mismatch or have been skipped. For more details consult the BAM file format description B.2 in the appendix.
8.4.2 Non-indexed aligner

The non indexed aligner is a very basic aligner, that was chosen because of its simplicity and then implemented in both Java and POP-Java. The reason to develop only this alignment algorithm on both POP-Java and Java is because of how the project developed and how the initial focus of the project was.

Figure 20 shows how the algorithm described in 6.3.1 was implemented. The class diagram only shows the pure minimum of the algorithm, not showing all the classes belonging to framework or to libraries used to load and save data.

The AlignCustom class is the controller of the algorithm that then creates different workers of the type AlignerCustomJob. When initializing the workers, they receive all the data needed with the setData function. When every worker is setup, they are started as a thread in the Java version, or the asynchronous method run is called in the POP-Java version.

The sequences get aligned in a very easy way. The following code shows how this is done for a single sequence.

```java
... char[] refBuffer = reference; char[] read = sequence.toCharArray(); int maxError = (int)(read.length * maxErrors); // test every position for(int i = 0; i < refBuffer.length - read.length; i++){
  int error = 0;

  // check if that position contains the sequence
  for(int loop = 0; loop < read.length && error <= maxError ; loop++){
    if(read[loop] != refBuffer[i+loop]){
      error++;
    }
  }

  if(error <= maxError){
    return i;
  }
}
...```

We see how the code simply iterates over the reference sequence while trying to find a position
where the sequence fits with less than the maximum errors allowed.

The POP-Java implementation uses the same class diagram, and the code is also nearly exactly the same. This shows the power of POP-Java, because it is easy to port a threaded implementation of an algorithm to a POP-Java implementation, that works the same way, but is also able to be distributed over the network.
8.5 Uniqueness map

The uniqueness map generation offers the user a way to identify the regions in a reference sequence that have many copies in the same reference. A region's uniqueness has a great influence on the ability of an aligner to align sequences to that region, simply because he cannot know to which one of those similar regions a sequence belongs. During this project, a uniqueness map generator has been developed, you can find the algorithm used in the algorithm section of this report [6.3.3].

Ideally, a uniqueness map is created with the same aligner that was used to align the actual sequences. But this is usually not possible because of performance reasons. If for example bowtie was used to align certain sequences to a reference sequence, and the bowtie would be used to create the uniqueness map, every possible sequence would have to be saved into a file and then aligned by bowtie, to be then mapped back to the initial location. While this works and is doable, the faster approach of creating a custom algorithm that is specialized on the creation of the uniqueness map was favored.

The simplified class diagram used to implement the algorithm can be found in figure [21].

```
stop -= partLength;
int refLength = ref.length - partLength;
int length = stop - start;
int dupes;
int wrong;

// Loop through all assigned positions
for (int base = start; base < stop && !stopJob; base++){
    dupes = 0;
    // Check all subsequent positions for duplicates
    for (int i = base + 1; i < refLength; i++){
        wrong = 0;
        // Compare base sequence to potential duplicate sequence
        for (int loop = 0; loop < partLength && wrong <= MAX_WRONG; loop++){
            // Implementation details...
        }
    }
}
```

Figure 21: Uniqueness map creation class diagram

The CustomUniqueness class is the controller of the algorithm. It will then create different workers, the CustomUniquenessJob classes. These workers have a reference back to the controller. Every worker gets a certain region of the reference sequence that he needs to cover. Whenever a worker finds a duplicate region, it uses the addValue function of the controller to increment the duplicate counter at both places in the uniqueness map, the source region and its duplicate.

The core part of the algorithm is the detection of the duplicates. The following code shows how this was implemented.
if (ref[i+loop] == '−'){
    wrong = MAX_WRONG + 1;
} else {
    if(ref[i+loop] != ref[base+loop]){
        wrong++;
    }
}

if (wrong <= MAX_WRONG){
    controller.addValue(i, partLength, 1);
    dupes++;
}

if (dupes > 0){
    controller.addValue(base, partLength, dupes);
}

The algorithm is constructed out of 3 for loops. The first loop increments a pointer to the start of the region we try to find its duplicates. The second for loop increments a counter that points to the start of the potential regions that could be duplicates of that region. With the third for loop those two regions are then compared. For every duplicate region we find, we send a signal to the controller to save the duplicate to a file. After all all possible regions with duplicates for a specific region were checked, the controller gets informed about the number of duplicates for the original region.
8.6 Variant detector

The variant detector created for this project is here to replace Varscan in case it is not installed. The output format of the custom varscan detector is identical to the Varscan output format.

For the creation of this code, much of the code used for the viewer application could be reused. The viewer application already checks on the fly the variations for the currently loaded region. The underlying code could be used and expanded to traverse a whole BAM file and export the results in the Varscan data format. The algorithm works the following way: The BAM file that has to be scanned is cut up into pieces of 10'000 bases. The BAM file is then queried for all sequences that overlap those regions. Using those sequences, for each base the total reads of all base types is calculated. The scanner then goes through every position and checks if there are any reads that are not equal to the reference sequence. If this is the case and the amount of reads is higher than the specified minimum amount of reads, the variant is saved into a file.

There are certain limitations with the implementation, while it does detect insertions and deletions, it is not as precise when reporting insertions or deletions that are more than one base long, for this task Varscan does a better job. The limitations are not technical but there was not enough time during this project to make those improvements. The code would not need to be rewritten, but could be extended to allow the detection of insertions and deletions with a bigger precision.

8.7 Data encoding / compression

A lot of data has to be transmitted during the whole process. Especially during the phases that run on the grid, the size of this data is critical. An interesting observation can be made that most data used in genetic analysis is all but encoded in a efficient manner.

For example, DNA consists of 4 bases, A, C, T and G. Sometimes N is also used to identify a base that is unknown but that is known to exist. Surprisingly reference sequences are usually encoded in standard ASCII or even UTF-16. This means for each base 1-2 bytes are used. This generates a big amount of data.

Often this data is then zipped. But this has some disadvantages. For example to access the data, the whole file needs to be decoded. DNA is also in general not very repetitive and because of this, hard to compress. What a compression algorithm will do, is not much more than to re-encode the data with a more efficient encoding. For the performance critical files, a better performing fileformat was searched for, or created.

8.7.1 Fasta reference

As already explained, Fasta reference files cannot be compressed by their nature to much more than simple recoding. A simple zip file could be used for our purpose, but that would have one disadvantage, random access to the file. Because we want to reuse the same fileformat as much as possible, and because the viewer should be able to load data over a network without loading the whole file, a zip file is not ideal. A simple encoding would be better because then the offset in the file can be calculated for each position.

By default, fasta files have a 1 byte per base encoding. We only need 5 characters, A,C,T,G and N. The idea is to encode a certain number of bases into one byte. To calculate the amount possible
is easy. One byte can hold 256 numbers. We just need to solve the equation:

\[ 5^x \leq 256 \]

\[ x = \frac{\ln(256)}{\ln(5)} \]

\[ x = 3.44 \]

Having 3.44 characters encoded in 8 bit is not possible, because of this we use 3 as the amount of bases we encode in a byte. This gives us a compression ratio of 3 over the standard fasta file. A quick test shows us that this encoding is indeed efficient. For the test the human chromosome 21 was used.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fasta</td>
<td>46.8MB</td>
<td>0%</td>
</tr>
<tr>
<td>Fasta-GZIP</td>
<td>11MB</td>
<td>76.5%</td>
</tr>
<tr>
<td>Binary</td>
<td>15.3MB</td>
<td>67.6%</td>
</tr>
<tr>
<td>Binary-GZIP</td>
<td>8.1MB</td>
<td>82.7%</td>
</tr>
</tbody>
</table>

The Binary file is about one third bigger than a GZIP file. But it is interesting to see that the compressed binary file is 25% smaller than the normal GZIP compressed file. Sadly because of the dependency on external applications that need fasta files as source files, we cannot transform all the fasta files to our binary format. That being said, the viewer application supports the binary fasta files, and a simple conversion application has been written. If needed, any fasta file can be converted to the binary format and be used with the viewer. This might be interesting when the fasta files are located on a remote server.

### 8.7.2 Aligned Sequence

For the aligned sequences, BAM, the binary SAM fileformat [14] has been used. It is an industry standard and open source libraries, like picard [26], exist to work with it in different programming languages. Over SAM, the BAM fileformat has a compression factor between 4 and 10.
9 Visualization Implementation

While the visualization of the processed data started with a good basis, several improvements needed to be made to the codebase. The most important and interesting changes are detailed in the following sections.

9.1 Resource caching

So that the user is able to work offline, the resources from web-resources need to be cached. Those resources can come from BioMart[9] or any other external resource. To handle this, a caching class has been developed through which the requests go. Each request is then identified by a hash, that is calculated depending on the information requested, usually the URL of the requested resource. Afterwards a file with the name of the hash is searched for in the cache folder of the application. If present, the cached file will be returned, if not, the requested resource is downloaded to the cache folder and then returned. The caching is optional and not enabled for all resources. Every remote resource could be cached, but it might be impractical to cache every big file.

In figure 22 we can see the general workflow. When the resource is requested, it is either directly returned from the cache if present, or downloaded, cached and then returned.

![Cache Sequence diagram](image)

Figure 22: Cache Sequence diagram

The caching obviously not only improves the fact that it is possible to work offline once a resource has been accessed once online, but the speed of the application is also improved greatly because we can load data directly from the local harddisk instead of requesting it on a remote, maybe slow server. Another advantage is that the server has less load to handle. For example, at each start of the application, the list of all known genes on the currently visualized chromosome is downloaded. If cached locally, this improves performance and serverload.
9.2 Paired-end visualization

More and more often the sequences that are aligned are paired-end reads. This means, for every sequences that is read, there is a second sequence that belongs to it. The advantage of this is that it is easier to map, because we know the approximative distance between both pairs.

During the semester project this alignment information was not shown during visualization. Now paired end reads are shown graphically with a line that connects both pairs. The result can be seen in figure 23.

![Paired ending visualization](image)

Figure 23: Paired ending visualization
9.3 Various improvements

There have been several other improvements throughout the visualization tool. For example, the visualization of the uniqueness map created by the framework can be seen in figure 24 or the coverage graph also created by the framework in figure 25.

Figure 24: Uniqueness map of a reference sequence

Figure 25: Coverage graph

Other small improvements are the ability to copy the content of a single sequence when selecting it, the display of the application license and numerous other changes.
9.4 Variant reporting

One important part of a genetic analysis is the final step of saving the results to be able to have a trace of the work done. There are several ways to save this information. Three approaches have been done in this project. The submission to a scientific webportal, the saving to the local framework and the saving into a local application independent html file.

9.4.1 Café Rouge

Café rouge is a webservice that intends to solve one common problem in bioinformatics. Various laboratories around the world search for variations in their patients. They do this for diagnostic or research purposes. The problem is that there is yet no standard way for those laboratories to interact with each other so that they can share the variants they found and the diseases they are associated with. Café Rouge tries to solve this problem by offering a framework that allows the laboratories to submit the variants with various information attached to them and publish them on a single repository. A user of the website can then for example subscribe to a RSS feed of a certain gene and be informed about new variants that are found on this gene.

The submissions on this website are anonymous for the patients involved. They are only identified by a user ID. If a laboratory wants more information about that patient, they can contact the submitter of the variant directly. This solves another problem with certain other services where you can store variants, but there is no way to know more about the variant because all the information that links it to a patient is lost.

Café rouge uses the VarioML XML format \[7\] for the variant submission. VarioML allows the detailed description of a Variant. The following listing shows an example file submitted to Café Rouge.

```
<variant type="cDNA">
    <gene source="HGNC" accession="COL1A1" />
    <ref_seq accession="NG_007400.1" />
    <name scheme="HGVS">c.579 delT</name>
    <genetic_origin term="paternal">
        <evidence_code term="inferred"/>
    </genetic_origin>

    <comment>
        <text>Variant inherited from affected father</text>
    </comment>
</variant>
```

We can see that one variant is submitted. The variant was found in the gene COL1A1 at position 579. The base T was deleted at that place. Further information and a comment is also provided. As mentioned, this is a minimal example of an entry.

9.4.2 Framework

The user has the possibility to save the variants found. They will then be saved into a XML file that be used by the framework. When an alignment is visualized, the visualizer will use this information to show which variants were already saved, helping to identify variants detected in other patients. Figure \[26\] shows how those variants are displayed inside the framework.

The user can directly visualize one of the variants through this list.

The ability to save the framework is useful because it allows the user to see if a variant was
9.4.3 HTML report

Having a paper copy of an analysis done on a patient is very useful. For this reason the possibility to export a report into the HTML format has been added. Those HTML reports can then either be printed, or stored in an application independent way. The result of such an export can be seen in figure 27.

![Variant report](image)

**Variant report**

**Subject**

ID: P1234  Name: Test  Gender: Female

**Information**

**Disease:** Cancer  
**Comment:** Several variants that might be related to the cancer this patient has. Further analysis has to be done.

**Variants**

<table>
<thead>
<tr>
<th>Chromosome</th>
<th>Gene</th>
<th>Pos</th>
<th>Variant</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>BRCA1</td>
<td>38451210</td>
<td>A/A</td>
<td>0.2652</td>
</tr>
<tr>
<td>17</td>
<td>BRCA1</td>
<td>38451219</td>
<td>G/G</td>
<td>0.2353</td>
</tr>
<tr>
<td>17</td>
<td>BRCA1</td>
<td>38451234</td>
<td>A/W</td>
<td>0.3529</td>
</tr>
<tr>
<td>17</td>
<td>BRCA1</td>
<td>38451237</td>
<td>T/T</td>
<td>0.29410000000000003</td>
</tr>
<tr>
<td>17</td>
<td>BRCA1</td>
<td>38451238</td>
<td>G/K</td>
<td>0.3529</td>
</tr>
</tbody>
</table>

**Author**

Author: Beat Wolf  Email: wolf@fryx.ch

![Variant report as a html document](image)
9.5 Annotation display

Several resources on the web provide annotation files for different features of the human genome, similarly several tools exist that output their results as annotation files. There are mainly two types of annotation files, Wiggle and BED files. Wiggle files are used for graphs, while BED files are used to draw regions, for example genes on a chromosome. One of the drawbacks of the viewer was that it did not allow to visualize those annotations. During this project, the different annotation formats were implemented. Currently the viewer supports the Wiggle data format, the BED data format, and the GFF data format which closely resembles the BED data format. It is possible to load the annotations dynamically after the application was started, but it is also possible to define which annotation will be used before opening the visualization. Figure 28 has an example of a wiggle file while figure 29 has an example of a BED file.

![Figure 28: Example of a wiggle file](image)

![Figure 29: Bed annotation file](image)
10 Performance tests

The main objective of those performance tests will be to compare the different implementations between each other. We want to see what the overhead is to use the pop technology and what speed increase we can expect when running the application on a grid. To perform these tests, two computers were used. The first computer was a Lenovo laptop with an Intel Core2Duo 2.4 GHz processor with two cores and 4 GB RAM. The second computer was a desktop computer with an AMD Phenom X6 1090 3.2GHZ processor with 6 cores and 8 GB RAM. Both computers were running Kubuntu 10.10 64 bit as their operating system. An important note to add is that the AMD processor has a technology called “Turbo boost” [27] which overclocks up to 3 cores if only 3 cores of the CPU are used. The overclocking is not done by a software and cannot be deactivated easily. The clock speed changes from 3.2Ghz to 3.6Ghz, which is a significant change in the CPU performance and might affect certain measurements. Both computers were connected to a 1Gb/s local area network. All the data used for the measurements can be found on the enclosed CD-ROM A.

As a first step, the three technologies, POP-Java, POP-C++ and pure Java are compared in a basic test application. To perform this test, the Pop example of a distributed Integer class is used. All three implementations need to perform the exact same job. The code of the main class that calls the different functions of the Integer class, written in POP-Java can be found below:

```java
public class TestInteger{
    public static void main(String [] args){
        Integer i1 = new Integer();
        Integer i2 = new Integer();
        i1.set(11);
        i2.set(14);
        System.out.println("i1="+i1.get());
        System.out.println("i2="+i2.get());
        i1.add(i2);
        System.out.println("i1+i2="+i1.get());
    }
}
```
Now, all different combinations were tested. A pure POP-Java implementation, a pure POP-C++ implementation, a POP-Java implementation where the Integer object is written in POP-C++, a POP-C++ implementation where the Integer object is written in POP-Java and a pure Java version. The following table shows how long it takes to execute the example. Every program was executed 10 times where the average was taken. No rules were specified in the POP framework that would restrict where the placement of the objects on a machine. The code, except for the Java code, was taken from the POP-Java and POP-C++ examples.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP-Java</td>
<td>7.03</td>
</tr>
<tr>
<td>POP-C++</td>
<td>0.18</td>
</tr>
<tr>
<td>POP-Java &amp; POP-C++</td>
<td>6.93</td>
</tr>
<tr>
<td>POP-C++ &amp; POP-Java</td>
<td>0.63</td>
</tr>
<tr>
<td>Java</td>
<td>0.13</td>
</tr>
</tbody>
</table>

The result is interesting. The example is so basic, it should not take much more time than loading the binary into the system memory should probably take most time. The POP-C++ seems to do exactly that. The pure POP-Java application on the other hand is 39 times slower to startup. While it was expected that the POP-Java implementation would be slower than the POP-C++ implementation, it was surprising to see that the combination of POP-Java and POP-C++ was so much slower than the other way around. This indicates a problem in POP-Java in the object creation. We will see the effects on this in further benchmarks.
10.1 Aligner

There are several aligners integrated into the framework, custom coded ones and external ones. In this chapter we try to compare the performance of the different aligners. To make the comparison more interesting, the aligners are separated into indexed and non indexed ones.

10.1.1 Indexed aligners

The indexed aligners use an index of the reference sequence to align the sequences faster. The included aligners are the bowtie aligner [21], the BWA aligner [23], the stampy aligner [24] and the custom indexed aligner. It is hard to compare those aligners because they do not all have the same features, and the end result varies between the different aligners. A comparison between them is nevertheless interesting.

As the dataset, 905'184 sequences were aligned against the human chromosome 17 which is 153 million bases long. The average length of the aligned sequences was 170 bases. The benchmark included the alignment of the sequences with the result of a sorted and indexed BAM file. Some aligners needed additional steps to convert their output to this specific result. The benchmarks have been done using the desktop computer with a 6 core processor. All aligners were run in their default configuration to have a fair comparison. The Stampy aligner could not be tested in this matchup, because it does not support multithreaded execution. Using a single thread, it required 7208 seconds (2 hours) to align the data of this dataset. The following table shows the result of this benchmark.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Bowtie</th>
<th>BWA</th>
<th>Custom indexed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>166s</td>
<td>160s</td>
<td>47s</td>
</tr>
<tr>
<td>2</td>
<td>108s</td>
<td>109s</td>
<td>28s</td>
</tr>
<tr>
<td>3</td>
<td>86s</td>
<td>93s</td>
<td>22s</td>
</tr>
<tr>
<td>4</td>
<td>74s</td>
<td>84s</td>
<td>20s</td>
</tr>
<tr>
<td>5</td>
<td>67s</td>
<td>79s</td>
<td>18s</td>
</tr>
<tr>
<td>6</td>
<td>64s</td>
<td>77s</td>
<td>17s</td>
</tr>
</tbody>
</table>

Figure 31: Indexed alignment speed comparison

In figure 31 we can see that consistently the indexed aligner developed during this project was faster than the other aligners. At first this might seem surprising, the aligner developed during this project did not receive nearly as much developer time as the other aligners, and it is written in Java, which does not necessarily have to be slow, but well written and compiled C/C++ code should be able to run faster. That being said, the custom algorithm does not align the sequences as good as for example BWA, which supports insertions or deletions in the sequences compared to the reference sequence.
To better judge the performance of those aligners when scaling over multiple processors cores, we look at the speedup graph in figure 32.

![Speedup Graph](image)

**Figure 32: Indexed alignment speedup**

Here it becomes apparent that the overhead of non multithreaded work that needs to be done is very important, as we can see that all three aligners do not scale so good over multiple cores as expected. To further investigate this, the two aligners that need to convert their output to a indexed BAM file are looked at more closely. The Bowtie aligner allows to output an alignment as a SAM file. This SAM file then needs to be converted to a BAM file, which then needs to be sorted and indexed. The BWA aligner does not allow to directly output a SAM file, but creates its own fileformat. The BWA aligner offers a utility to convert this custom file format into a SAM file. Afterwards the samtools are used in the same way as with the Bowtie aligner. All those conversion steps and even the sorting step are not multithreaded. In figure 33 we can see the different work steps decomposed for Bowtie and BWA, using 6 cores, with the same dataset as before.
It is directly apparent that both aligners lose much time with the data conversion which is not multithreaded. The custom developed aligner does not have this problem. Because it uses the Picard\textsuperscript{25} library, it is possible to write directly into a sorted and indexed BAM file, not using an intermediate file format. Looking at those numbers, the raw performance of the custom aligner seems much less impressive, but still very competitive, certainly compared to bowtie that produces a very similar alignment output. We can see the benefit of a well integrated algorithm, that allows to create the data directly in the format that is needed by the application.
10.1.2 Non indexed aligners

There are only two non indexed aligners, both coded during this project. Both use the same underlying algorithm, described in the algorithm chapter [6.3] The comparison between them is interesting because one uses normal Java Threads, and the other uses POP-Java to implement the parallelization, even across the network. When running both implementations on the desktop computer using the same dataset, aligning 1788 reads on the BRCA1 gene which has 81,116 bases, we could see some interesting results [34]

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java, 2 threads</td>
<td>26s</td>
</tr>
<tr>
<td>POP-Java, 2 workers</td>
<td>43s</td>
</tr>
</tbody>
</table>

![Figure 34: Non indexed aligner speed](Image)

The POP-Java implementation used 2 worker objects, the Java implementation 2 Threads. Both only used the local computer for their calculations. Any differences in speed cannot come from a network latency. Further analysis of the performance revealed the following interesting fact. The setup time of the 2 POP-Java workers needs 11 seconds, while the setup time of the two Java threads, which includes the data loading that they need, is only 1.5 seconds. Accounting for this setup time, we can see that in this setup, the POP-Java implementation is 23% slower than the pure Java implementation. Some of that time might come from the communication overhead between the POP-Java objects. For every sequence that is successfully aligned, 1 function call is made that needs communication between two POP-Java objects. For this dataset, this are 1920 calls.
The next step to analyze the performance of the algorithm and POP-Java was to test how well both implementations scaled over multiple cores. The test was performed using both computers at disposal. At first, the cores 1-6 were used on the desktop computer, and afterwards, the POP-Java implementation scaled further to the 2 cores of the laptop. The following table shows the result of this measurement.

<table>
<thead>
<tr>
<th>Cores</th>
<th>POP-Java(total)</th>
<th>POP-Java(setup)</th>
<th>POP-Java(working time)</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>67s</td>
<td>10s</td>
<td>57s</td>
<td>48s</td>
</tr>
<tr>
<td>2</td>
<td>43s</td>
<td>11s</td>
<td>32s</td>
<td>26s</td>
</tr>
<tr>
<td>3</td>
<td>36s</td>
<td>13s</td>
<td>23s</td>
<td>18s</td>
</tr>
<tr>
<td>4</td>
<td>35s</td>
<td>16s</td>
<td>19s</td>
<td>15s</td>
</tr>
<tr>
<td>5</td>
<td>35s</td>
<td>19s</td>
<td>16s</td>
<td>12s</td>
</tr>
<tr>
<td>6</td>
<td>37s</td>
<td>23s</td>
<td>14s</td>
<td>11s</td>
</tr>
<tr>
<td>7</td>
<td>40s</td>
<td>27s</td>
<td>13s</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>44s</td>
<td>31s</td>
<td>13s</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 35 shows the results in a graphical way.

As we can see in figure 35, the startup time for the POP-Java objects is a very important part of the total execution time. The startup time is composed by the time it takes to setup the POP-Java objects and send them over the network. Most of the gain that new workers bring in terms of lowering the time needed to make calculations, is removed by the overhead of the POP-Java object initialization. In that particular setup, the optimal amount of workers is between 4 and 5. Afterwards the overhead of POP-Java cannot be compensated by the speed increase of the calculation. But what is actually interesting to see is that the progression of the actual working time of the POP-Java implementation, is very similar to the Java implementation. To better see the actual benefit of adding new cores to the different implementations, we take a look at the speedup graph.
We can see that the pure Java implementation scales very well over multiple cores, whereas the POP-Java version is really hurt by the overhead of the object creation.
10.2 Variant scan

Two options are present in the framework to scan an alignment for variants. The custom developed variant scanner, and the Varscan variant scanner. Both scanner behave in a similar way and for that reason it was interesting to see how they perform against each other. Two data sets have been used to test the two implementations. First a small dataset, using 1788 reads, aligned to the BRCA1 gene which has 81’116 bases. Depending on the analysis, this is a very common dataset. The second dataset was a large one using 4’284’126 reads aligned on the chromosome X which has 153 million bases. All tests where repeated 5 times and the average of the measurements where taken.

<table>
<thead>
<tr>
<th>BRCA1, 1788 reads</th>
<th>Chromosome X, 4’284’126 reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom</td>
<td>Custom</td>
</tr>
<tr>
<td>0.4s</td>
<td>2.7s</td>
</tr>
<tr>
<td>Varscan</td>
<td>Varscan</td>
</tr>
<tr>
<td>200s</td>
<td>170s</td>
</tr>
</tbody>
</table>

Figure 37: Integer test execution times

It is interesting to see how both implementation perform in a similar way. While the custom algorithm is faster for smaller datasets, the Varscan is faster for large datasets. Possible optimizations for the custom algorithm would be to improve the way how the aligned BAM file is accessed. To access the BAM file the Picard [26] is used. One of the biggest problems is that the Picard library does not allow to know where the different aligned sequences are, but requires to query the file for every region. Finding a solution for this problem could improve the performance very much.
10.3 Uniqueness map

For the uniqueness map creation, one algorithm was implemented with two technologies as described in the Algorithm chapter [6.3]. Because of the slow speed of the algorithm, at first tests were only done on a single gene, the BRCA1 gene which has only 81'116 bases, and not on a full chromosome. The scaling of the two implementations over multiple cores was benchmarked. For the normal Java version, up to 6 cores on the desktop computer were used. The POP-Java version scaled over both available computers, up to 8 cores in total. Again for those tests, the “Turbo boost” [27] that overclocks the CPU if only 3 cores are used needs to be kept in mind when analyzing the results. All measurements were repeated two times and their average was taken. We can see the result of this benchmark in the following table and figure [38].

<table>
<thead>
<tr>
<th>Cores</th>
<th>POP-Java(total)</th>
<th>POP-Java(setup)</th>
<th>POP-Java(working time)</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27s</td>
<td>4s</td>
<td>23s</td>
<td>31s</td>
</tr>
<tr>
<td>2</td>
<td>22s</td>
<td>7s</td>
<td>15s</td>
<td>16s</td>
</tr>
<tr>
<td>3</td>
<td>18s</td>
<td>10s</td>
<td>8s</td>
<td>11s</td>
</tr>
<tr>
<td>4</td>
<td>20s</td>
<td>13s</td>
<td>7s</td>
<td>8s</td>
</tr>
<tr>
<td>5</td>
<td>22s</td>
<td>16s</td>
<td>6s</td>
<td>6.6s</td>
</tr>
<tr>
<td>6</td>
<td>25s</td>
<td>20s</td>
<td>5s</td>
<td>5.8s</td>
</tr>
<tr>
<td>7</td>
<td>28s</td>
<td>23s</td>
<td>5s</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>31s</td>
<td>27s</td>
<td>4s</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 38: Uniqueness scaling over different threads

There are different interesting observations that can be made from those measurements. The most obvious one is the high overhead of POP-JAVA compared to the actual calculations that need to be made. On the other hand, we can see the similar progression of the actual calculation time in the POP-Java implementation to the Java implementation, which shows that a part from the POP-Java Overhead needed for the startup, the POP-Java implementation scales actually very well over multiple cores and computers. The most surprising result, at first, is that especially when using only one core, the POP-Java implementation is much faster than the normal Java implementation. The reason for this is actually very simple. If we look at figure [13] we see that the saving of the duplications that were found is done in the controller object of the algorithm. In the Java implementation, even though all workers work in a different thread, when they call a function in the controller object, this function is executed in the same thread as the worker calling it. This is different in the POP-Java implementation. There the worker calls a asynchronous function on the controller, which is then executed on a different core than the worker which can continue to work during this time.

Looking at the speedup factor of both implementations scaling over the different scores [39] we can see the impact of the overhead that POP-Java creates very clearly.
While the Java implementation scales very well over the different cores, the POP-Java implementation, which uses the same underlying algorithm, hits a maximum very soon.
Because of the high overhead POP-Java created, a bigger example than the BRCA1 gene was used for further tests. For this reason, parts of the DMD gene were taken as the dataset. The DMD gene is the largest human gene with 2.5 million basepairs. Because of the high algorithmic complexity of the algorithm, only 170k bases were used, which is about the double from the previous dataset. The exact same tests as before have been performed.

<table>
<thead>
<tr>
<th>Cores</th>
<th>POP-Java(total)</th>
<th>POP-Java(setup)</th>
<th>POP-Java(working time)</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>112s</td>
<td>4s</td>
<td>108s</td>
<td>118s</td>
</tr>
<tr>
<td>2</td>
<td>65s</td>
<td>7.6s</td>
<td>57.4s</td>
<td>66s</td>
</tr>
<tr>
<td>3</td>
<td>52s</td>
<td>11s</td>
<td>41s</td>
<td>46s</td>
</tr>
<tr>
<td>4</td>
<td>47s</td>
<td>14s</td>
<td>33s</td>
<td>38s</td>
</tr>
<tr>
<td>5</td>
<td>44s</td>
<td>17s</td>
<td>27s</td>
<td>31s</td>
</tr>
<tr>
<td>6</td>
<td>44s</td>
<td>20s</td>
<td>24s</td>
<td>27s</td>
</tr>
<tr>
<td>7</td>
<td>44s</td>
<td>24s</td>
<td>20s</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>45s</td>
<td>27s</td>
<td>18s</td>
<td>-</td>
</tr>
</tbody>
</table>

As we can see, the overall picture is very much the same as in the previous tests, but with a much lower POP-Java compared to the actual calculation time. All other observations from the previous measurements are also valid for those tests. The lower overhead also has an impact on the speedup of both implementations as can be seen in figure 41.
10.4 Conclusion

One of the goals of this project was to analyze POP-Java and its usefulness in this domain. After having done different tests, we can see that POP-Java is indeed an interesting approach for grid-computing. From a performance point of view, no visible speed decrease could be seen coming from the communication between the objects. Currently the only major problem is that the setup time of a POP-Java object is quite slow, which removes its usefulness for calculations that are shorter than a few minutes. But we could show that the usefulness of POP-Java increases with the size of the calculation, and, by nature, grid-computing is used for resource intensive applications. Nevertheless, improving the POP-Java startup performance would be worth looking into.

As mentioned in 6.3.1, a multilayer distribution algorithm could be used, where POP-Java distributes the workload over the network to different machines, but locally normal Java threads would be created to distribute the load over the different processors/cores of the machine. This would greatly reduce the problem of the slow POP-Java object creation.
11 Results

In this chapter we take a closer look at the results that were achieved during this project. As a quick reminder, the goals of the project were the following:

- Creation of a framework allowing to do all the steps of genetic variant detection and analysis.
- Integration of grid-computing into the framework to improve performance
- Enhancing the visualization of the results

Those goals were all achieved. In the following sections we see the details of those 3 goals. Overall nearly 25'000 lines of code have been written during this project and the semester project leading to it.

11.0.1 Framework results

The framework allows to execute all the steps of a variant analysis. Here is a short list of the features implemented to analyse sequenced data.

- Import of FastQ or 454 Fasta/quality files
- Import of BAM/SAM files
- Ability to quality check and filter the imported data
- Sequence alignment using Bowtie, BWA, Stampy, a custom basic alignment algorithm and a custom indexed algorithm
- Variant analysis using varscan and a custom variant detector
- Alignment coverage calculation
- Uniqueness map calculation

The framework offers more than just to run those tools. The idea was also to make the work with the patient data easier. The following features were implemented to achieve this goal:

- Import of gene sequences directly from Ensembl
- Ability to manage different patients with their analysis
- Manage all known variants found in the different patients
- Proxy server support

The final look of the framework can be seen in figure 42. To have a better overview of the different functionalities implemented, the user manual can be consulted.
11.0.2 Distributed computing results

The goal for the distributed implementation was to see if it is possible at all to use the grid in our situation and what the benefits are. Different algorithms, namely an alignment and uniqueness map creation algorithm, were ported to POP-Java and executed in a grid environment. The results showed that the use of the grid environment indeed works and can improve the performance. The analysis showed that the current implementation of POP-Java is mainly useful for large calculations, and could be optimized for smaller calculations by improving the setup speed of POP-Java objects. The combination of POP-Java and POP-C++ could not be tested because of current limitations of those frameworks.

Testing the usefulness of distributed computing for this particular project, but also to test the POP-Java framework in a practical environment. During this project, several bugs could be found, and test-cases produced to reproduce them. Thanks to this, many of those bugs could be solved and the framework became more useful.

11.0.3 Visualization results

The improvements on the visualization application were mainly to bring it to a state where it can be used in production. Many bugs were fixed, and new features were implemented. Here is a list of the features that were implemented during this project.

- Report creation as a HTML file
- Report variants to Café Rouge
- Display of uniqueness and coverage maps
- Dynamic import of annotation files
- Ability to load a binary reference file
• Overall optimizations and stability improvements

The main window of the final visualization application can be seen in figure 43.

Figure 43: Visualization window
12 Future improvements

Many features could be completed and work well, but there are still areas where the application can be improved. While all features are implemented, some polishing needs to be done to transform the framework into an application that can be used on a day to day basis.

A big part of the improvements that need to be done are bug fixing and polishing, but there are also other parts that could be improved in the future. In this chapter we look at a list of possible future improvements.

12.1 Faster uniqueness map creation

The creation of the uniqueness map is currently very slow. Using the same technique as the custom indexed aligner, which means using a index of the reference to find possible alignments, would improve the performance very much. The current algorithm works for smaller genes, but it is problematic for larger ones.

12.2 Multilayer parallelization

The POP-Java algorithms could be reimplemented using a different approach, where POP-Java would be used for the network distribution, but Java threads would be used to distribute the workload locally on the different processors of the machine.

12.3 Align indels

While the custom indexed aligner works reasonably well and fast, the major drawback is that it cannot align deletions and insertions. The usefulness of the algorithm would be greatly improved if this was implemented. Several alignment algorithms exist [25] that could be implemented using, as the starting point, the possible locations identified by the index.

12.4 Command line tools

Having the custom tools well integrated into the framework is very useful, but it would be very interesting to have the tools available as stand-alone command line tools. That way, in case the framework does not exactly answer the needs of a specific situation, the tools could still be used, for example on a server with not graphical user interface.

12.5 User testing

One big aspect that needs to be done to create a good application that is usable in a real life environment is user testing. During the project, except for the two conferences visited, there were not many possibilities to have a real user that usually works as a geneticist, to test the application. Several techniques can be used to do the user testing. My recommendation is to create a list of tasks the user needs to perform. While the user tests the application, he should be observed, but do not helped, unless he gets stuck in a way that he cannot finish the task. During the observation, notes should be taken.

12.6 Better archiving

While the user can save all the files associated with an entry in the framework, it would be much easier if he could create an archive for a certain patient, that could then be easily reimported into the framework later.
12.7 Speed improvements

Several parts of the application still have room for optimization. For example, the file-format in which the uniqueness map is stored, while very small and fast to access, does have its shortcomings when the application needs to read large amounts of it. For example, if the user is zoomed out to a point where each pixel on the screen represents 5000 bases, every single base needs to be read. It would be more efficient to save for example the average of groups of 1000 bases, so that if zoomed out, not every base needs to be read, but only the averages.

Other improvements that could be made would be to create index files for different file-formats that are used. For example the varscan file format which is a pure text format, cannot be accessed in an efficient manner because the different entries do not have the same length. Other file-formats have the same problem mainly the annotation files.

12.8 Various improvements

There are several more improvements that can be made, to make the framework more useful.

- Integration of more external tools
- Better control over the external tools
- Better feedback, like console output, of the external tools
- Adhere to the HGVS\textsuperscript{33} standard for the description of sequence variants
- Ability to use transcript based variant positions

13 POP-Java conclusion

During the development, the distributed computing integration was done using the POP technology, and mainly by the POP-Java implementation of this technology. While the final code works well, several problems were encountered during the development, some of them were fixed, and some yet need to be addressed. First a list of the solved problems:

- Impossibility to use packages with POP-Java
- Impossibility to add a classpath to the POP-Java compilation
- Inability to begin lines with a space, but with any other character works
- Inability to create arrays of POP objects
- Inability to use packages when mixing POP-C++ and POP-Java
- Impossibility to use POP-Java objects on any other computer than localhost
- Inability to pass a char array from a POP-Java object to a POP-C++ object
- Inability to save callback objects as a variable of a object, but need to be used as parameters

While those problems could be corrected, several others still remain. Here is a list of those problems.

- Sequential methods work the same way as concurrent methods
- Limited to one main method in all POP-Java classes (two different Java files cannot have both a main method)
• Impossibility to use POP classes with their whole identifier. Example: pop.MyClass

• Impossibility to pass a POP-Java object reference to a POP-C++ object

• Slow POP-Java object creation

• POP-Java objects with circular references are not destroyed correctly

Some of those limitations could be worked around, and some are bigger problems. For example the fact that sequential methods execute the same way as concurrent methods seriously limits the ability to program in the way the POP model is intended and it forces the use of semaphores that otherwise would not be necessary. The slowness of POP-Java object creation seriously limits its use for smaller calculations where the overhead for the POP-Java environment became quickly more important than the actual calculations that had to be performed. The inability to use a combination of POP-Java and POP-C++ objects in a callback setup, where the controller is written in POP-Java and the workers that use the controller object as a callback are written in POP-C++, made it impossible to test the performance of a POP-C++ worker compared to a POP-Java worker.

Some of the already corrected problems were fixed quite late during this project, which made it difficult to make full use of the POP technology. Nevertheless, the integration of POP-Java in the project to distribute the workload was a success and works very well. After the most important bugs mentioned will be corrected, POP-Java will be a very nice to use framework for distributed computing.

14 Conclusion

During this project, a framework was developed that helps a researcher or a doctor to analyze genetic data. Different tools were integrated into the framework, allowing the user to perform the whole process from the acquisition of the raw genetic data coming out of a DNA sequencer, up to the final diagnostic. To perform this work, the user does not need special technical skills, but is guided through the process with an intuitive interface. The project simplifies the currently common process of running several tools manually, and having to worry about the sometimes incompatible output formats those tools have. Also the value of the previously developed visualization application was greatly improved by the integration into the framework, while still keeping the ability to run as a stand alone application.

The integration of POP-Java to explore the ability to run certain computer resource intense tasks in a distributed environment was also a success that showed the potential of the POP-Java technology. Through this project, the POP-Java technology has been brought to a much more mature state that makes it useful in real projects. The ability to use distributed technologies also enables new business models that are interesting to explore even more in the future.

The project was very interesting and showed the power of computer science and how it is used in all domains of the current society. As a computer scientist I was very interested to apply what I learned to a concrete problem. Not only did I learn much about computer science during this project, but also about genetics. I’m especially glad to have had the opportunity to visit two scientific conferences during this project which allowed me to talk directly to people working in the domain, and see what the current challenges are in genetics.

Starting the master project with an introduction through a semester project was great, as I did not have to learn about the subject at the beginning of the project, but was actually confronted with the problems the framework tries to solve through the work I did earlier. The project was a success and I am looking forward to implement the missing features after the master project by working directly for Phenosystems SA.
References

[1] European seventh Framework Programme
http://cordis.europa.eu/fp7/

http://www.gen2phen.org

[3] Gen2Phen grant

[4] NMD-Chip
http://www.nmd-chip.eu

[5] NMD-Chip FP7 grand

[6] Café Rouge
http://www.gen2phen.org/groups/cafe-rouge-development

[7] VarioML

[8] Ensembl
http://www.ensembl.org/

[9] BioMart
http://www.biomart.org/

[10] GAPSS
http://www.lgtc.nl/GAPSS/

http://galaxy.psu.edu/

[12] Cloudburst
http://cloudburst-bio.sourceforge.net

[13] Digipede
http://www.digipede.net

[14] SAM/BAM Fileformat
http://samtools.sourceforge.net

[15] POP-C++
http://gridgroup.hefr.ch/popc/

[16] Eclipse IDE
http://www.eclipse.org

[17] JUnit
http://www.junit.org

[18] JUnit Benchmark
http://labs.carrotsearch.com/junit-benchmarks.html

[19] Samtools
http://samtools.sourceforge.net
Analysis and visualization of DNA sequences using cloud computing

[20] Varscan
http://varscan.sourceforge.net

[21] Bowtie
http://bowtie-bio.sourceforge.net

[22] Burrows-Wheeler-Transformation

[23] BWA
http://bio-bwa.sourceforge.net

[24] Stampy
http://www.well.ox.ac.uk/project-stampy

[25] Alignment algorithms
http://en.wikipedia.org/wiki/Sequence_alignment

[26] Picard Java library
http://picard.sourceforge.net

[27] AMD Turbo Core
http://de.wikipedia.org/wiki/AMD_Turbo_Core

[28] How to Build a Better DNA Search Engine
http://www.technologyreview.com/blog/arxiv/25371/

[29] Git versioning system
http://git-scm.com/

[30] EGit eclipse plugin
http://www.eclipse.org/egit/

[31] Visualization of DNA sequence coverage
Enclosed CD-ROM

[32] Hadoop framework
http://hadoop.apache.org/

[33] HGVS naming standard
http://www.hgvs.org/mutnomen/recs-DNA.html
A CD-ROM content

The CD-ROM enclosed with this report contains all the data needed for this project. The content is the following:

- Application executable
- Documentation
- Documentation source (LaTeX)
- Source code
- Javadoc of the source code
- Sample data
- Benchmark data
- POP-Java documentation
- “Visualization of DNA sequence coverage” documentation

B Data formats

While not all data formats are described here, you can find the most important data formats with a quick overview, and resources of where you can find more detailed information about them.

B.1 Varscan

The varscan data format is a commonly used data format to store variants. It is a simple text file where the data is organized with tabs. For each variant different information is stored. Here is the list of the saved data:

- Chromosome in which the variant has been found
- Position on the chromosome of the variant
- Reference sequence at this position
- The actual variant sequence
- The Total amount of reads for the reference sequence
- Total amount of reads for this variant
- Frequency of the variant
- Strand on which the variant was found (forward or backward strand)
- Quality readings
- P value of the variant.

Here you can see an example of a varscan file:

<table>
<thead>
<tr>
<th>Chrom</th>
<th>Position</th>
<th>Ref</th>
<th>Var</th>
<th>Reads1</th>
<th>Reads2</th>
<th>VarFreq</th>
<th>Strands1</th>
<th>Strands2</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>19054</td>
<td>G</td>
<td>C</td>
<td>11</td>
<td>2</td>
<td>15.38%</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

More information about the data format can be found on the official website: http://varscan.sourceforge.net/using-varscan.html
B.2 BAM

The BAM data file format comes together with the SAM file format. The BAM file format is the binary version of the SAM file format which is used in this project. The BAM files are used to save alignments of sequences. The BAM file format is too complex to make a simple description that would be useful. It is advised to consult the official data format description at http://samtools.sourceforge.net/SAM1.pdf

B.3 Wiggle

The Wiggle data format is used for graph annotations. It is a pure text based data format, allowing for 2 types of graph annotations. First there is the basic graph type where for each point in the graph, the value is given. You can see here an example of such a data file:

```
track type=wiggle_0 name="variableStep" description="variableStep_format" visibility=full autoScale=off viewLimits=0.0:25.0 color=50,150,255 yLineMark=11.76 yLineOnOff=on priority=10
variableStep chrom=chr19 span=150
59304701 10.0
59304901 12.5
59305401 15.0
59305601 17.5
59305901 20.0
```

The second type of graphs defines the graph in a different way. Only the starting position and the span of each value is defined. That allows to only save the actual values on the graph without their positions.

```
track type=wiggle_0 name="fixedStep" description="fixedStep_format" visibility=full autoScale=off viewLimits=0:1000 color=0,200,100 maxHeightPixels=100:50:20 graphType=points priority=20
fixedStep chrom=chr19 start=59307401 step=300 span=200
1000
900
800
700
600
```

You can find more details about those data formats on the official website: http://genome.ucsc.edu/goldenPath/help/wiggle.html

B.4 BED

The bed file data format is used to store region based annotation. Different regions can be defined in this text based dataformat, and the way how those regions will be drawn can be defined. The following information is defined for each entry:

- Chromosome
- Starting position
- End position
- Name
- Score, which defines the color of the feature
- Strand on which the annotation is
• Different visual information, like colors

Here you see an example of this data format:

```
track name=pairedReads description="Clone Paired Reads" useScore=1
chr22 1000 5000 cloneA 960 + 1000 5000 0 2 567,488, 0,3512
chr22 2000 6000 cloneB 900 − 2000 6000 0 2 433,399, 0,3601
```

website: http://genome.ucsc.edu/FAQ/FAQformat#format1

### B.5 GFF

The GFF file format has the same goal as the BED file format. It contains the following information about each annotation:

• Chromosome name
• The source of the information
• The name of this annotation
• The start and the end of the annotation
• The score of the annotation
• The strand on which the feature is

You can see an example file here:

```
track name=regulatory description="TeleGene (tm) Regulatory Regions"
chr22 TeleGene enhancer 1000000 1001000 500 + . touch1
chr22 TeleGene promoter 1010000 1010100 900 + . touch1
chr22 TeleGene promoter 1020000 1020000 800 − . touch2
```

You can find a detailed format description on the official website: http://genome.ucsc.edu/FAQ/FAQformat#format3

### B.6 Fasta

The fasta file format is used to store sequences. This can be a reference sequence, or sequences that come out of a sequencer. It is a very basic text based fileformat. It consists of a header for each sequence that starts with the “>” character, followed on the next line the actual sequence of bases. Here you can see a simple example of a fasta file:

```
>Sequence 1
ACGTGTCTTATTGCCATCGATCGGCATCGACTGACGACATGACATTCATTTTTGTGGTACGATCAACGAT
ATCGCTGCGATAGATATGCATCGACATCATTTCATCAAACCCGGACACTACGAGCAGCTAGCTAGC
```

More information can be found on the wikipedia page about this fileformat: http://en.wikipedia.org/wiki/FASTA_format

### B.7 FastQ

The FastQ fileformat resembles the fasta fileformat. It is used to store sequences, but alongside quality information. It is the standard file output format of the Illumina sequencer. Every sequence starts with a header file beginning with a @ followed by the sequence name. The next line contains the different bases of the sequence. After that a line starting with a + can be found. After that a line containing the quality of each base of the sequence can be found. The quality encoding either starts at the value 33 or 64 depending on the encoding used. The value 33 is equivalent to the ASCII character ! and the value 64 to the character @.
Analysis and visualization of DNA sequences using cloud computing

@SEQ_ID
GATTTOGGTTCAAAGCAGTATCGATCAATAGTAAATGCATTGTTCAACTACAGTTT
+
! ' ' (((((((+++)%%%++)(%%%%)1+++—+++'))))**55CCF>>>>>>CCCCCCC65

The detailed description of the file format can be found on the following Wikipedia page:
http://en.wikipedia.org/wiki/FASTQ_format
C Manual

In this manual, the basic usage of the framework and the viewer application are explained.

C.1 Installation & Updates

To install the Framework, you need to unpack the framework archive which can be found on the enclosed CD-ROM. In the extracted folder, you can execute the framework.bat batch file if you are using Windows, or the framework.sh file if you are using Linux.

The procedure to upgrade an existing installation is very easy, all you have to do is to replace the framework.jar file from the old release with the framework.jar file from the new release. This works well for smaller releases. For bigger releases you also might want to replace the tools folder. Alternatively you can unpack the new release into a new folder, and move all files with a .xml extension from the folder of the older release, into the newly created folder. That way you migrate all your data to the latest release.

C.2 Framework configuration

There are several configuration options that can be done inside the Framework. The main configurations are the location of external tools, and the information about the user of the application, which will be reused when submitting variant reports.

To access the configuration options, click in the File menu on Configuration. The first tab, figure 44(a), will let you configure the location of the external tools. If a external tool has been configured correctly, its name will be marked in green.

The second tab, as seen in figure 44(b), allows the configuration of the user information. When submitting a variant report, this information will be used so that you don’t have to type it in with every new report.

The last tab, figure 44(c), allows you to configure a proxy if there is a need for it in your network.

![Configuration dialog](image)

Figure 44: Configuration dialog

The fully use the framework, references against which you want to work need to be defined. To open the reference manager, open the File menu and click on “Manage references”. The reference manager will open.
You can now either manually add new references through the input on the lower part of the dialog, or import genes from the ensembl website with the import button on the bottom right of the dialog. To import your own reference, you need to specify the species of the reference, for example Human, the version of the reference, for example hg19, the chromosome, and the position of the reference on this chromosome. If you import a whole chromosome as a reference, specify 0 as the position. As the file you need to specify a valid Fasta file containing the reference.

If you wish to import a gene from ensembl, you will see the dialog in figure 46. In this dialog you can choose between hg18 and hg19 as the reference version of the human genome, and the different chromosomes. After selecting the chromosome number, the list on the bottom will populate with all the genes known to ensembl for this chromosome. The input field lets you filter the genes to select the one you want to import. After selecting the gene you want to import, use the import button on the bottom. The gene will the appear as a reference in the reference manager.

C.3 Patient management

The framework allows the management of different patients. Every patient has a name, and ID and a gender. You can easily create new patients either over the File menu using the “New Patient” entry, or by right clicking into the patient explorer. The new patients are then added to the patient explorer view on the left side of the application.
To rename a patient, or delete it, all you need to do is right click on the patient, the popup menu will present you with the needed options.

Figure 47: Patient explorer

Figure 48: Patient explorer popup
C.4 Data importing

The framework allows you to import raw sequence reads, or prealigned sequence files. To import a raw sequence files, open the File menu and click on the “Import raw data” entry. The importer wizard will open. On the first page of the wizard, figure 49, you can choose the file type you want to import. Either you import the traditional fastq file containing all reads with their quality readings, or you choose the 454 option, where you need to specify the two different files, the sequence file and the quality file. After choosing the files you want, click on the next button.

![Figure 49: Raw data importer start](image)

After checking the fastq file to import, or converting the 454 files into the fastq data format, a overview of the quality of the fastq file is given. You can easily check the number of sequences, and their lengths. That way you can judge the quality of your read, before doing the whole analysis. After verifying the quality, you can continue by clicking the next button.

![Figure 50: Raw data importer quality control](image)

The next page in the wizard, see figure 51, gives you the possibility to filter the sequences, depending on certain tags. You can add as many tags as you want to the list, every sequence will be checked if it starts with this tag. If it does, the tag is removed and the sequence is put in the group of this tag. All sequences in the same group can be saved to a patient in the final step of the import. You are not required to add any tag.

The next page, shown in figure 52, lets you define some filters, for example the minimal or maximal length of a read. You can also specify that each read needs to be trimmed, this means that a certain
amount of bases are cut of at the beginning or end of the read. This can be used if the quality of the readings is bad at the end of the read. After defining the filters, the button next will apply them.

On the last page of the wizard you can select to which patient you want to add each group of reads. The different groups are defined by the tags you added in the preprocess step. After selecting where the reads are saved to, click on the finish button. If you wish to discard a certain group of reads, select the user “IGNORE” for those reads.

You can now see your read as a subgroup of your patient. Clicking on the read will give you new options, for example to align it against a reference sequence.

If you already have aligned your read in an external application and want to import it, you can do that. Choose in the File menu the “Import alignment” entry. The importer lets you choose a name for the imported alignment, the file that contains the alignment, the patient you want to associate with it and the reference against which the alignment was made. Make sure that you choose the same reference as you used when you aligned the alignment externally. The specified file has to be a sorted and indexed BAM file, or a SAM file. The SAM file will be sorted and aligned while importing.

After the import is done, you can find the alignment in the patient you selected. The read will have the name “Imported”, having a alignment named the way you configured it during import.
C.5 Sequence alignment

The next step after importing the data into the framework, is aligning it to a reference sequence. To do this, make sure that at least one reference sequence was configured in the reference manager. Afterwards select your read. The main view of the application will now show the selected read. See figure 55. You have the possibility to change the source FastQ file if needed, and an experimental support for paired end reads is present, where you can specify the second FastQ file, representing the paired read. The most important part is that you can choose the reference against which you want to align the sequence. If multiple references are selected, multiple alignments are made.

After selecting your reference sequence, you can select the aligner of your choice and run the alignment. Keep in mind that only the aligners that are configured correctly will be shown. If the aligner you are searching for is missing, check in the framework configuration if the path to the executable is correct. After launching the alignment you will be presented by a dialog that lets you choose some configuration options for the aligner. Not all aligners offer such options. Figure 56 shows you an example of such a dialog, with the options that the bowtie aligner proposes.

A progress bar will appear in the right part of the application, indicating the advancement of the alignment. After the alignment is finished, you can find it as a sub element of the read.

C.6 Data analysis

While it is possible to visualize a newly aligned alignment directly, it is more interesting if some data analysis is done before. The current options are to calculate a coverage graph, which will be used inside the visualization tool, find all variants in the alignment, this allows you to quickly navigate to interesting variants when you visualize the data, and the alignment simulation, which creates a uniqueness graph of the reference sequence that can be shown alongside the rest of the data during visualization.
Figure 55: Raw read display

Figure 56: Bowtie aligner options
To calculate the coverage all you need to do is click on the coverage button in the alignment view \[\text{Alignment display}\]. After the process is finished, you can see that it worked by looking at the number displayed to the left of the button, indicating the maximum coverage in this alignment.

Scan for variants is just as simple, but this time you have the option between different types of variants scanners. Choose the one you like and run it. You will be presented with a dialog that lets you choose the options for the variant detection \[\text{Varscan options}\]. You can set the minimum amount of coverage at the place of a variant for it to be accepted, the minimum overall frequency of the variant and the minimum number of reads that had that particular variant. After running the variant scan you will be able to use them directly in the visualization.

The last analysis you can do is the alignment simulation. To do this, again from the main alignment view \[\text{Alignment display}\] you select the size of the simulation pieces. During the simulation, the reference will be cut up into pieces, those pieces are then mapped back to the reference. That way we get uniqueness map. The size of those pieces can be configured. If a uniqueness map was already calculated for a reference sequence with the same piece length, it will be marked in green, this means you don’t have to recalculate the map.
Be aware that this calculation takes a long time.
C.7  Data visualization

The final visualization of the data is a big part of the application. It allows you to visualize the alignment, the different variants that have been found, the uniqueness map of the reference, gene annotations and several other information. In this manual we try to go through this different features.

C.7.1  Navigation

There are several ways to navigate through a loaded file. You can zoom, scroll and jump. These feature can all be found in the status panel at the bottom of the application. In figure 59 you can see the scroll to navigate left right in the application. Under the scroll you can see from the left to the right:

A zoom input field, where 100 is the closest zooming level and 1 the highest.
A zoom slider, that lets you scroll with the mouse, the input field is updated accordingly.
A input field to jump to a certain position, it also shows the current position. If no base is selected, the left most base is used as the position.
A jump button that lets you jump the the position entered into the position field.

![Figure 59: Status bar to navigate](image)

Additionally different keyboard shortcuts can be used to navigate the genome. To use them, a base needs to be selected. The left and right arrow key will move the currently selected base. The CTRL key combined with the left or right arrow will move the selection to the next or previous page. It is also possible to use the scrollwheel of the mouse on the sequence panel.
C.7.2 Information

When navigating the genome, different information can be inspected. This information can be found in the main view, or in the sidepanel of the application.

First there is the quality graph. By default it shows the estimated quality of the base shown in the colored sequence bar. The higher the bar, the better the quality.

Then there is the base sequence, and the sequence that is visualized.

In the read sequence, you can see colored bases, where the color is an indication for the quality of the read. Green means a good quality and red a bad quality. A mutation is shown in a distinct red color.

You can copy the currently visible sequence to the clipboard. This is done in the menu under File / copy region or with the CTRL + c command.

![Figure 60: Information display](image)

It is possible to change the quality graph to a graph that shows the amounts of reads that each base had line in figure 61.

On the top left you can see the scale, the number represents the number of reads that a bar has that goes up to the top.

To get this view, chose it in the menu under View / Quality viewer / Amount.

The colors used for the different bases are: A = green, C = blue, T = red, G = black.

![Figure 61: Quality viewer](image)
The actual data that we visualize, the individual sequences of a particular read, is in the central part of the application. Each rectangle represents one sequence, that contains several bases. The sequences are color coded to represent their quality. The darker the color, the better the quality. The individual bases have the same color coding as the sequence viewer.

![Sequence visualization](image)

Figure 62: Sequence visualization

Further information can be found on the right side information panel. You can find the general information about the current genome which the species it comes from, the chromosome used and the total length, see figure 63.

![General information](image)

Figure 63: General information

It is also possible to select a single base to get more details about it. You do this by clicking on the base. You will then be presented with the detailed information in the right side panel. You will be able to see the individual reads of each base and it’s position like in figure 64.

![Base information](image)

Figure 64: Base information

You can do the same for an individual sequences. Selecting one will highlight it and present you with the information about its size and quality.

Once a base or a sequence is selected, you can unselect it by clicking on an empty spot on the sequence viewer.
C.7.3 Genes

If you are browsing the alignment of a chromosome, you might be interested to restrict your analysis on a certain gene. There are two ways to have more information about the genes. First you can get a overview of all genes of the chromosome and where they are located. To see all the genes in the visualization tool, select the “Genes” view in the “Views / Graphs view” menu. All the genes of the chromosome will be shown, and if close enough you can even see the names.

While this view is interesting to see which gene affects a certain region of the chromosome, it is also interesting to have all the information about a genes and its transcripts. To do this, open the gene selector in the “Navigation” menu under “select gene”. In figure you can see the dialog that will open up. You can filter the list of all genes to find the one you search and select the gene by either double clicking, or using right click to open its context menu. The currently selected gene is marked in green.

After selecting the gene you want, it all of its transcripts will be displayed. The blue transcripts are protein coding transcripts, the red parts of them are their actual coding regions. To know more about a transcript you can right click on it. The right click menu also allows you to access the configuration about which transcripts should be visible and which not.
C.7.4 Variations

The main source of interest while doing the analysis are the different variants that can be found in an alignment. If before visualizing the data a variant detection has been made, you will be able to visualize all variants by clicking on “Search variant” in the “Navigation” menu. The dialog from figure 69 will be presented to you.

As you can see, all variants are displayed in a long list. While it is interesting to sort that list, for example after the frequency of the variants, it is even more useful to use the filter system at the bottom of the dialog. Using the different properties of the variants and simple “equals” “bigger than” and “smaller than” rules, you can filter out the variants you want.

Keep in mind that when selecting a gene, only the variants that are actually on that gene will be shown. Also the column “Consequences” will only be used if a gene is selected. This column tells you the consequences of a variant on that gene. The worst consequence of all its transcripts is displayed.
C.7.5 Report generation

After all the work was done, you probably want to create a report of your findings. Currently there are three possibilities to save your results. You can save them into the framework, for later usage with other patients, you can save them as a html report or you can submit your findings to Café Rouge [6]. For all three possibilities you need to select the variants you want to report. Either right click on a variant you find interesting in the main view 70(a) or mark it using the variant explorer 70(b).

After marking all variants you want to report, open the “Report” menu and choose the “Report variants” entry. It will open a dialog 70 where you can add some additional information to the report. If you configured the framework correctly, all the information should already be set, for example your name, or the patient identity.

![Figure 70: Variant report dialog](image)

Before creating the report, you need to select the variants you actually want to have in the report from the list you created at the top of the dialog. After selecting those variants, you can click on the “Prepare report” button to finalize the report. After clicking on the prepare report button you will be presented with the dialog from figure 71.

On this last dialog you can set the specific details for your details, like the disease the variants
Figure 71: Variant report finalization

are connected to and a comment. You get the choice between saving the report, which will save the variants to the framework, exporting to Cafe Rouge, which will send the data directly to Café Rouge, or exporting to a html file, which will create a html file of the report.
D Planning

<table>
<thead>
<tr>
<th>Weeks</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Viewer improvements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gen2Phen conference</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NMD Chip conference</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project definition and prototyping</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interface ideas and partial prototype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tasks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Archiving analysis and prototyping</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer/Holidays</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel algorithms analysis and prototyping</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finalizing / finishing features</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 72: Planning of the master project