Parallelized FPGA-based Graph Creation for De Novo Genome Assembly
Leveraging the Reconfigurable High-Performance Platform RIVYERA

Supervisor: Dipl.-Inf. Lars Wienbrandt
Author: Ulf Rüegg
Docent: Prof. Dr. Manfred Schimmler
Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

______________________________
Abstract

In bioinformatics, complex algorithms are necessary to assemble short fragments of DNA data into the original sequence – the genome of an organism. This task, also referred to as de novo genome assembly, is very data-intensive and time-consuming, especially for large genomes (e.g. the human genome). Most well-known assemblers divide the overall task into smaller steps, forming an assembly pipeline.

To cope with both, data and time requirements, an approach is presented that makes use of the massively parallelized FPGA-based hardware platform RIVYERA. A well-known data structure – the de Bruijn graph – is adapted and brought into an optimized form, fitting the architecture of RIVYERA. Evaluations indicate a significantly decrease of the execution time required to build the graph, roughly ten times faster than PASHA, one of the fastest state-of-the-art assemblers. Thus, this work proofs the suitability of RIVYERA for de novo genome assembly and presents a data structure that is required by later steps of the assembly pipeline.

Keywords: Bioinformatics, DNA, De Bruijn Graph, De Novo Genome Assembly, FPGAs, RIVYERA
# Contents

1 **Introduction** 1  
  1.1 Purpose of this Thesis ........................................... 3  
  1.2 Outline .......................................................... 4  

2 **Basics of Bioinformatics** 7  
  2.1 Biological Background ............................................. 7  
  2.2 Genome Sequencing ................................................... 11  
  2.3 Genome Assembly ....................................................... 12  
  2.4 Assembly Strategies ................................................... 17  
  2.5 Assembler Comparison .................................................. 25  

3 **RIVYERA** 29  
  3.1 FPGAs ................................................................. 29  
  3.2 Architecture ........................................................... 32  
  3.3 Communication API ..................................................... 33  

4 **De Novo Genome Assembly on RIVYERA** 37  
  4.1 Concept and Data Structure .......................................... 37  
  4.2 Implementation ......................................................... 50  

5 **Comparison and Evaluation** 61  
  5.1 Performance Evaluation ............................................... 75  

6 **Conclusion** 79  
  6.1 Future Work ........................................................... 80  

Detailed Tables of Content 84  

Bibliography 93
In 1990, the Human Genome Project [SSHP93] was initiated. It aimed to collect the overall DNA information of the human genome, including the identification of all human genes. Officially, the project completed in 2003. However, even today there are still uncertainties left [Bak12]. The Genome 10K project [Gen09] desires to provide a collection of the genomes of ten thousand vertebrate species. Both projects involved and still involve many scientists and require lots of effort and funding. The Genomes Online Database (GOLD)\(^1\) [PLJ+12] holds over 20,000 projects aiming to sequence a genome of an organism (as of February 2013). What justifies this effort and these high costs?

With the information of the human DNA biologists are able to determine causes for disorders and diseases. Likewise, ten thousand genomes of different vertebrates allow deep insights into evolution, amongst other fields of interest.

To retrieve the DNA information of a genome, a complex biological process has to be performed in which the DNA is cut into pieces and then is puzzled back together. This task is impossible without the help of modern technologies and computers that help to tame the complexity and the large amounts of data emerging during this process. By way of illustration, the human genome consists of roughly 3 billion units of information that have to be composed out of small chunks. Each chunk contains less than one thousand pieces of information - a puzzle of more than 1 million pieces without a priori information of the assembly.

**Genome Sequencing and Assembly**

In the late 1970s Frederick Sanger developed a method to extract the sequence of bases from a piece of DNA [San77] – referred to as *genome sequencing*. It was the first technology in this area, and Sanger received the Nobel prize for his research in 1980. Loosely speaking, the sequence of the four building blocks of the DNA (adenine, cytosine, guanine, and thymine) is converted into a computer readable sequence of the four letters A,C,G, and T. From a technical point of view, it is not possible to retrieve the overall sequence of an organism’s DNA at once. For this reason, small chunks of the overall DNA are created and sequenced.

\(^1\)http://www.genomesonline.org/
1 Introduction

separately, each resulting subsequence is referred to as read. Nowadays, several so-called second-generation sequencing technologies are available. They produce larger amounts of data in fewer time and at lower costs. However, the outputs of Sanger sequencers are up to a length of 1000 bases [PLA09], as opposed to reads originating from second-generation sequencing are most of the time shorter than 100 bases [Mar08].

Genome assembly describes the process to re-assemble the reads into their original sequence. This process requires sophisticated algorithms and lots of computing resources. State-of-the-art assemblers run either on supercomputers or on large computing clusters. For instance, Li et al. [LZR+10] required a supercomputer with eight 2.3 GHz quad-core CPUs and 512 GB working memory to assemble a human genome in roughly 44 hours. However, first effort is being made to perform assemblies, even for large genomes, using mediocre computing resources [YMC+12].

Several difficulties exist that genome assembly has to overcome. Compared to the overall genome size, reads are very small (as mentioned above, the human genome contains about 3.2 billion bases). Furthermore, the sequencing process introduces errors in the reads. This might introduce faulty connections as the possible overlaps between any two reads are the only hints that an assembly algorithm can use. Such errors have to be identified and removed during the assembly process.

An important question considers the quality of an assembled genome. Plenty of metrics exist, for instance, assemblers compare themselves in terms of the maximal and mean length of joined reads. Sometimes used metrics are not very expressive to determine the quality of an assembly and sometimes they are even misleading. In any case, newly assembled genomes require lots of polishing and post-processing until a certain quality can be guaranteed [Bak12].

Every year several new approaches to genome assembly are published improving genome assembly in terms of execution times, requirements of computation resources, and quality.

Present Difficulties Current state-of-the-art assemblers face the problem that second-generation sequencing technologies produce immense numbers of short read data. Processing this data takes time, e.g., for human-sized genomes the latest assemblers run about a day and more [LZR+10, LSM11]. Due to the shortness of the reads, the de Bruijn graph became the de-facto standard data structure for de novo genome assemblers. In this data structure, reads are split into even smaller subsequences of a read, so-called k-mers. Consequently, the large amount of data and the fact that read data is error-prone yields memory requirements for human-sized genomes that cannot be fulfilled by standard purpose computers. For instance, Li et al. require 140 GB of runtime memory for the assembly of the human genome [LZR+10]. As a result, the latest de novo
1.1 Purpose of this Thesis

Assemblers depend on computing platforms with high acquisition and operating costs.

To reduce the memory requirements most assemblers use algorithms to remove or correct erroneous reads before the de Bruijn graph is constructed. These algorithms make decisions based on the counts of read subsequences and are referred to as error-correction algorithms. The sole counting of subsequences reduces the memory demands, but still every read needs to be process possibly multiple times. For this reason, pre-assembly error-correction algorithms are one of the most time-consuming steps of the overall assembly process. For some assemblers error-correction can consume up to half of the overall execution time [LZR+10].

Massively Parallelized Computing

State-of-the-art assemblers exploit problem-inherent parallelism to improve execution times. The term massively parallelized computing refers to a concept where a large number of computing units is used to solve a task collaboratively. Such a proceeding pays off especially for problems that are not solvable by general purpose computers within a reasonable amount of time. As an example one can consider cracking a password. A naive approach is to try all possible character combinations that can make up the password. A standard computer might require decades to try each possible password while specialized platforms can solve this problem in less than a week [KPP+06] (depending on the encryption technique).

A specific massively parallelized platform is RIVYERA, which heavily relies on FPGAs. FPGAs are integrated circuits that are programmed using a hardware description language and allow to be reconfigured to diverging needs. Thus, highly optimized problem-specific solutions can be developed, while allowing a high level of parallelism and customized pipelining. Additionally to the FPGA itself, RIVYERA adds another level of parallelism by connecting multiple FPGAs via a bus system (a typical number is 128 FPGAs). A so-called host PC is able to exchange data with all available FPGAs using a PCIe bus.

RIVYERA’s predecessor COPACOBANA proved the strengths of this approach by performing a brute-force attack on the DES encryption algorithm in less than a week on average, while latest standard PCs still require decades [KNPR08]. Several algorithms of bioinformatics were adapted to RIVYERA and show significant speedups [WBB+10]. Therefore, it seems natural to evaluate RIVYERA’s capabilities in connection with de novo genome assembly.

1.1 Purpose of this Thesis

The goal of this thesis is to realize one of the most time consuming steps of the overall genome assembly. Several assemblers use a data structure called de Bruijn graph. This data structure has to be constructed using the special
1 Introduction

purpose hardware RIVYERA. The aim is to leverage its parallelism capabilities to accelerate the time-consuming problem of the genome assembly. Genome assembly includes several complex steps but the efforts here are limited to the de Bruijn graph creation.

1. A proper representation of the data structure has to be found matching FPGAs’ capabilities. Languages used to describe an FPGA’s design do not offer any data structures with a high abstraction level ad hoc, for instance, tree structures. Possibilities for parallelism have to be identified and utilized.

2. The graph creation is just a part of the whole genome assembly. Implications on following steps, e.g., originating from the selected data structure, have to be considered and evaluated.

3. A program has to be created that realizes the chosen approach. The program must integrate seamlessly into a future pipeline that is able to perform the overall genome assembly. Tests and evaluations are required.

Contributions

All previously stated problems are solved. A proper data structure is presented exploiting the characteristics of genomic data and forming a memory hierarchy to allow quick access. This structure represents a distributed de Bruijn graph which contains all information necessary to perform further graph processing steps, as done by related assembly algorithms. A reference implementation is presented and used to perform extensive evaluations.

1.2 Outline

The thesis is structured as follows. Chapter 2 presents the biological background required to understand all parts of this work. The terms of genome sequencing and genome assembly are explained in further detail and state-of-the-art technologies are discussed. Furthermore, assembler comparisons conducted by different authors are presented. Chapter 3 introduces the massively parallelized hardware platform RIVYERA and gives a short insight into its usage paradigms.

Chapter 4 presents the main contribution of this thesis. The concepts of the chosen data structure and its usage are illustrated, the resulting implementation is explained in detail. Chapter 5 evaluates the presented approach in several terms, e.g., execution-time and work-load distribution among different FPGAs.

Finally, Chapter 6 sums up the thesis and draws conclusions with relation to what has been achieved and what improvements can still be made.
1.2 Outline
Basics of Bioinformatics

This chapter introduces basic terminology and concepts of bioinformatics. First, the biological background is explained, terms such as DNA, gene, and protein are explained shortly. A summary of the technologies of genome sequencing and genome assembly is given. Several genome assembly algorithms are presented in further detail, especially with relation to their specific usage of the de Bruijn graph. Last, the latest evaluations and comparisons of the existing assemblers are outlined.

2.1 Biological Background

All living organisms are composed out of at least one cell and are differentiated as prokaryotes and eukaryotes. Eukaryotes’ cells have a core called nucleus. The nucleus contains chromosomes, of which the total represents the DNA information of an organism, also referred to as the genome [Nai07]. For prokaryotes, the DNA information is contained in the cell itself.

If each chromosome exists twice, a genome is called diploid, haploid if only one chromosome exists. For diploid genomes it is possible that the same chromosome exists with diverging bases at some locations. This is referred to as heterozygosity.

The DNA itself has no active functionality, it is simply an information carrier. This information is copied during a process called replication, e.g., for cell division, and used for the production of proteins, a process called gene expression [Sam12]. Gene expression is the basis for all life and utilizes the information of a gene to synthesize, for instance, proteins. The exact procedure is discussed in further detail shortly.

To technically retrieve the information of a genome, the DNA has to be extracted from the cell, cut into small pieces, multiplied, and read. This process, known as sequencing, yields large sets of fragments, much smaller than the original DNA strand. These fragments have to be put back into their original order, known as assembly, to obtain the actual genome information [Zer09].

As an example, the human DNA is comprised of 23 chromosomes, varying in length from 48 million bases to 249 million bases. The overall genome consists
Figure 2.1: Illustration of the DNA double helix and its components, the bases. G and C, and A and T are complementary of each other and connected by a hydrogen bond. Figure by Samuelsson [Sam12].

of 3.2 billion base pairs, of which only 0.2% differ between distinct individuals. However, numerical 0.2% are still 6.4 million base pairs [Nai07].

**DNA** The Desoxyribonucleic Acid (DNA) consist of a so-called backbone which is a long chain of sugars and phosphates. *Bases* are connected to the sugars, encoding the actual information, as the sugar is always the same. Four different bases exist: *adenine*, *cytosine*, *guanine*, and *thymine*, abbreviated as A, C, G, and T. From a chemical point of view, A and G are *purines* forming a double ring, C and T are *pyrimidines* forming a single ring. See Figure 2.1 for an illustration. The combination of the three components sugar, phosphate and base is referred to as *nucleotide*, which in terms of information content, is equivalent to the plain base.

The two ends of a DNA strand are called 5’ and 3’ end due to different properties of the sugar at the respective position. Reading from either direction results different sequences, the DNA is said to have a *polarity*.

In 1953 Watson and Crick [WC53] discovered the double-stranded nature of the DNA forming a *double helix*. In this form one strand is always the opposed one of the other, they are called to be *reverse complements* of each other. As shown in Figure 2.1, an adenine base is always connected to a thymine base via a hydrogen bond and a cytosine base to a guanine base [Sam12].
2.1 Biological Background

**Genes and Proteins** *Genes* are subsequences within the DNA and only make up a very small fraction of the overall sequence. They hold the information how to produce a specific protein and heavily differ in length. Genes are as small as a thousand bases or up to a billion bases and are located in either strand of the double-helix.

*Proteins* are sequences of *amino acids*. Amino acids are distinguished as *standard* and *non-standard*. Each of the 20 standard amino acids is directly encoded in genes, see Figure 2.3 for a list. The non-standard amino acids emerge, for instance, from modifications to existing proteins.

Proteins are responsible to execute nearly every function of an organism’s cell. The specific function depends on the shape of the protein, which is directly determined by the actual sequence of the amino acids. Three examples of the performed tasks are passing messages between cells and forming antibodies or hormones. Some proteins are also called *enzymes*, which guide plenty of biological reactions [Sam12].

**Gene Expression** As mentioned, the information encoded in the DNA is used to produce proteins. This procedure is called *gene expression*, generally known as the *central dogma of molecular biology*. An illustration of the process of gene expression can be seen in Figure 2.2.

First, a Ribonucleic Acid (RNA) copy of the gene is made during *transcription*. RNA differentiates from DNA in its type of sugar in the backbone, in the replacement of thymine by *uracil*, and in the fact that it is single-stranded. The RNA copy of the DNA is also called messenger RNA (mRNA) and it is responsible for the creation of a protein. mRNA is created by the enzyme *polymerase*.

Genes are further divided into *exons*, which hold the actual encoding information, and *introns*, which hold no relevant information. Introns might contain old information that was discarded during evolution. After transcription, during the process of *splicing*, introns are removed from the mRNA, such that only the exons remain.

The mRNA sequence uniquely encodes a sequence of amino acids, where the unit of three bases, called a *codon*, encodes one specific amino acid. The rules, assigning codons to amino acids are shown in Figure 2.3, and are referred to as the *genetic code*. Since codons comprise of three bases, \(4^3 = 64\) permutations are possible to encode 20 amino acids. Despite the codon AUG, which represents the amino acid *methionine* and serves as a *start marker*, each amino acid is represented by at least two codons. This fact is also known as the *degeneracy* of the genetic code. Furthermore, three specific codons serve as *termination symbols*, namely UAA, UAG, and UGA.

**Relevance** Scientists study the impacts of mutations within the DNA of different individuals. During evolution plenty bases within the overall sequence change. If a base of a start codon changes, a whole gene will be discarded and the individual
is hindered to produce the corresponding protein. Amongst others, this might be the reason for a genetic disorder, which scientists seek to cure.

To do so, they require detailed knowledge about the DNA sequence itself, the differences between individuals, and the function of a certain base. For instance, by comparing the sequences of an ill and a healthy individual, it might be possible to identify the reason for a genetic disorder. The differences in the DNA strand of two individuals are also referred to as Single Nucleotide Polymorphisms (SNPs).

Often, not the whole genome is reassembled, as it is an expensive and time-consuming task, instead regions with verified SNPs are analyzed. Nevertheless, de novo assemblies are necessary to find SNPs in the first place. Additionally, variations are often discarded when assembling the DNA of an individual by using an existing reference genome. Besides, de novo assemblies are necessary to get a first draft of yet unknown genomes. As of January 2012 it is possible to assemble the genome of a human individual in one day and for only $1000 [Bak12].
2.2 Genome Sequencing

The term *genome sequencing* describes the process to digitalize the DNA information of a genome. The genome is broken up into smaller chunks which are processed using varying technologies resulting so-called *reads*. Basically, reads are sequences of the four characters A, C, G, and T, each representing one of the DNA’s bases. One genome is sequenced multiple times, yielding the *coverage* of a data set. Higher coverage allows better and more complete reconstructions of the whole DNA sequence. The coverage diverges with respect to different areas of the overall genome. Several regions have higher coverage, while some regions are not covered at all, although the probability for uncovered regions is kept as low as possible.

Different sequencing technologies are categorized as first generation, second or next generation, and third or next-next generation [STK10, Met10]. First
generation sequencing is mainly represented by Sanger Sequencing, which was developed by Sanger around 1975. It produces reads of length 500 to 1000 bases with a typical coverage of 10× \([SPZ^{+12}]\), which in terms of length is superior to current second and third generation sequencing. A drawback of Sanger sequencing is that it yields lower throughput and is expensive, e.g., the human genome took about 10 years to be fully sequenced. The Sanger sequencing process roughly works as follows and is illustrated in Figure 2.4a: First, multiple fragments of the same chunk of the sample DNA are created by using biological methods. These fragments all start from the same base but differ in length. Next, each fragment’s last base is colored, the color allows distinction of the four bases. Last, the fragments are sorted according to their length, allowing to infer the original DNA sequence by reading the last bases in the previously generated order [STK10].

Second generation technologies aimed to produce higher throughput while lowering costs. This is achieved by performing the sequencing task highly parallelized, with the negative effect of lowering the reads’ lengths. The lengths diverge between 30 and 400 bases, depending on the technology. Furthermore, reads are affected by higher error rates, especially at the rear part. The characteristic procedure for most second generation sequencing is a so-called wash-and-scan routine. A large amount of copies of the original DNA sample is produced in a preprocessing step, which introduces first errors. These samples are attached to a solid surface and chemical fluids are added. The fluids contain colored elements that attach to the bases of the samples. Different fluids are added to stop the attaching process, followed by a scanning of the colored elements to determine the bases for read creation. Afterwards, the marking elements have to be adapted for the next wash-and-scan cycle, see Figure 2.4b for an illustration [STK10]. Examples of second generation technologies are Roche’s 454, Illumina’s Genome Analyzer, and Life/APG’s SOLiD [Met10].

Third generation technologies are still evolving and follow a variety of different strategies. They aim for even higher throughput, shorter run-times, and fewer sample material consumption to lower the costs. Increased read lengths are desired to solve problems during genome assembly, which arose from the shortness of second generation read lengths [STK10].

### 2.3 Genome Assembly

The previously described process of genome sequencing yields large data sets of reads. These reads are of different length, quality, and type, e.g., fragment or mate-pair. Mate-pair reads, also called paired-end reads, are two associated reads for which the rough distance between each other in the original genome is known. Fragment reads do not provide such information. The reconstruction of a genome’s DNA sequence from a dataset of reads, without a reference genome, is
2.3 Genome Assembly

Figure 2.4: Sequencing technologies. (a) Traditional Sanger sequencing, terminating bases are labeled and subsequences ordered by length. (b) Shows the typical wash-and-scan cycle used by Illumina sequencing. Figure by Schadt et al. [STK10].
basically known as *de novo genome assembly*. It is a data- and time-intensive task and is performed by high-performance and special-purpose computers [MKS10].

Several different assembly strategies exist, which mostly base on the common assumption that overlapping reads presumably are located in the same area of the genome. All of the various strategies start with a provided data set of reads and use them to create so called *contigs*, longer sequences of bases that cannot be extended using only overlapping information of the original reads. The contigs are ordered into *scaffolds*, e. g., by using mate-pair information. Scaffolds also hold information about their orientation, i. e., from which DNA strand they are, and information about gaps that have to be inserted due to missing bases. Figure 2.5 presents a good overview of this process [MKS10, NM11].

Genome assembly algorithms have to cope with several difficulties. As mentioned earlier, already the sequencing procedure introduces errors, which have to be detected or corrected. The DNA of an organism might contain *repeats*, subsequences of bases that occur multiple times in the overall genome. Repeats that are longer than the read length are hard to distinguish, and possible sequencing errors within repeat areas increase this difficulty. Furthermore, due to the double stranded nature of DNA, it is not safe to say in which direction a genome fragment was read during sequencing. For this reason, the reverse complement of each read has to be considered as well. Another problem are palindromes, DNA sequences equal to their reverse complement. Assemblers using so-called k-mers (subsequences of a read of length $k$) can solve this problem by using an odd $k$, since palindromes are only possible for even $k$s [MKS10].

To determine the quality of a completed assembly, several metrics exist. Examples are weighted median statistics, such as N50, maximal lengths of contigs or scaffolds, and reference genome coverage [LSM11]. A detailed description of such metrics, as well as a high quality evaluation of several state-of-the-art assemblers, are provided by Earl et al. [EBS+11], Narzisi and Mishra [NM11], and Salzberg et al. [SPZ+12].

**Overview** Miller et al., as well as Narzisi and Mishra, differentiate three main approaches to genome assembly. All three of them base on graph structures: *Overlap graph* (also called *Overlap-Layout-Consensus*), *de Bruijn graph*, and *greedy* graph algorithms [MKS10]. Each of these approaches is discussed in the following.

**Overlap-Layout-Consensus** Overlap-Layout-Consensus (OLC) approaches are the classic choice for read data produced by Sanger sequencing. The rough idea is to represent reads as nodes and overlaps between the reads as edges. Within this graph a *hamiltonian path*, a path that traverses every node exactly once, needs to be found. As this problem is NP-hard, heuristics are applied [NM11].

Basically, three steps are performed. First, all reads have to be compared pairwise, resulting a certain score which indicates the potential overlap between the
Figure 2.5: An overview of the assembly pipeline used by SOAPdenovo. Most state-of-the-art assemblers make slight alterations within each step and use different representations of the data. However, in general their overall procedure is quite similar. Illustration by Li et al. [LZR+10].

(A) Starts with different types of reads from different sequencing platforms, e.g., fragment or mate-pair. (B) Represents the reads as a graph to allow traversals. (C) Perform error corrections, a broad variety of strategies exist. (D) Extract longer sequences from the graph and output contigs. (E) Use additional information, such as mate-pairs, to order and connect contigs into scaffolds. (F) Post-processing of the scaffolds, e.g., close gaps by using original read data.
two reads. Second, an overlap graph is created, where nodes represent the reads and edges are created between two nodes if their reads overlap. The calculated score for this overlap is associated with the edge. After further graph processing a rough arrangement of the reads is known. With additional information, e.g., the reconsideration of the original read data, the explicit read arrangement is determined and the final sequence is assembled [MKS10]. Well-known assemblers are CELERA [Mye00], CABOG [MDK+08], ARACHNE [BJK+02], Minimus [SDSP07], and Edena [HFF+08].

The OLC approach has become less important with the emerging of second generation sequencing technologies since such reads are too short to achieve sufficient assemblies. Furthermore, due to the large amount of produced reads, pair-wise read overlap tests are too expensive. However, according to Miller et al. increasing read lengths of third generation sequencing technologies lead back to overlap graphs [MKS10]. An up-to-date example is the String Graph Assembler (SGA) from Simpson and Durbin [SD12].

**Greedy** Assemblers that Miller et al. categorize as greedy algorithms base on either an internal representation using an overlap graph or a de Bruijn graph (see the next paragraph). They result from a formalism to find the shortest common superstring, which is an NP-complete problem [NM11]. To achieve reasonable execution times, a greedy procedure is applied. Within a set of reads, the overlap with the best score is picked, merged and added to the pool of reads again, until no further merging is possible. The remaining sequences are used as contigs for the next steps.

Narzisi and Mishra name a variation of greedy assemblers, called seed and extend. These assemblers use efficient prefix-trees, start with a seed and lengthen the seed in either direction until no further extension can be picked [NM11].

This kind of algorithm emerged with first results of second generation sequencing [MKS10]. Prominent examples are PCAP [HWA+03], Plusion [MN03], SSAKE [WSJH07], SHARC GS [DLBH07], VCAKE [JRB+07], Taipan [SSBSP09], and the PE-assembler [AS11].

**De Bruijn Graph** As mentioned, the OLC approach requires pair-wise read comparison to determine overlaps. This procedure is very time-consuming, especially for large sets of read data. A category of algorithms, that avoid this issue, uses a graph structure similar to de Bruijn graphs. In a de Bruijn graph redundant paths are implicitly stored in a compacted way. This is well suited for large volumes of short read data [MKS10]. De Bruijn graphs were introduced to bioinformatics by Pevzner et al. [PTW01], along with the EULER assembler as an exemplary implementation.

De Bruijn graphs use the notion of \( k \)-mers, which are arbitrary extracts from a read of length \( k \). In one form de Bruijn graphs consist of nodes, representing \( k \)-mers, and edges between two nodes, if the bases of two \( k \)-mers overlap in
2.4 Assembly Strategies

$k − 1$ bases. The graph is constructed by splitting each read of a data set into its consecutive k-mers displaced by one base. For two such k-mers an edge is introduced, see Figure 2.6 for an example. Some assemblers create edges only for k-mers occurring in the same read, other assemblers create edges for all existing k-mers of the used data set that overlap in $k − 1$ bases.

Finding the genome’s DNA sequence in a de Bruijn graph equals an eulerian path problem with the goal to traverse every edge in the graph exactly once. With error free read data this problem is solvable in linear space and time. But as mentioned above, read data contains a lot of errors, resulting in a graph that requires even more memory than those graphs used by the OLC approach. Furthermore, it is possible that multiple eulerian paths exist within the graph. Hence, an eulerian superpath has to be found, which in turn is an NP-hard problem, requiring the assemblers to apply heuristics [NM11].

Every de Bruijn graph-based assembler adapts the graph structure to its needs. Some assemblers weight the nodes, some weight the edges, whereby the weight equals to the frequency of a k-mer’s transitions to a specific base (see Figure 2.8a for an illustration). Often, an error correcting preprocessing step is performed prior to the creation of the graph. Single base errors in reads are detected and corrected or removed, for instance, by using counts of k-mers. This aims to reduce the size of the graph. Therefore memory requirements and computation time are decreased since erroneous branches are removed beforehand.

Paths through this graph represent possible contigs. Many of the paths are spurious since they result from k-mers introduced by sequencing errors. For instance, tips are introduced by an erroneous base at the rear end of a read, and result in dead ends of low frequency. Figure 2.7 shows a tip and two further common errors: bubbles, resulting from base errors in the middle of a read, and ambiguous links.

This thesis presents the construction of a specialized de Bruijn graph. For comparison, prominent examples using the de Bruijn graph approach are discussed in further detail in the next section.

2.4 Assembly Strategies

In the following, several strategies of de novo genome assemblers are presented. As this thesis focuses on the creation of a de Bruijn graph data structure used for error corrections and contig generation, the discussion of related assemblers addresses the kind of de Bruijn graph used. Procedures to remove errors, such as tip removal, are mentioned briefly. Assembling steps after contig generation, e.g., scaffolding, are discarded, as they are not relevant to this thesis.

**EULER – Pevzner et al. [PT01]**

Pevzner et al. were the first to use a de Bruijn graph for genome assembly in their **EULER** assembler. They adapted an approach of Indury and Waterman [IW95],
Figure 2.6: Example of a de Bruijn graph, where nodes represent k-mers (k=4) and transition weights count k-mer transitions within the read data. (a) shows the graph for a read ACGCGAT, (b) adds a read TCGCGAA.

who proposed to use a sequence graph for the assembly of the results of a sequencing technology called **Sequencing by Hybridization (SBH)**. SBH detects all subsequences of length \( k \) within a genome using biochemical techniques. These subsequences are also referred to as k-mers [Zer09]. The EULER assembler was designed for long Sanger reads and was extended by Chaisson and Pevzner [CP08] to support short reads as well (EULER-SR).

In a preprocessing step EULER-SR performs error corrections. Errors within sequenced reads are detected by identifying bases that occur with a low frequency compared to bases of a high frequency within the same subsequence. Such reads are either corrected or discarded.

A de Bruijn graph is constructed from k-mers, where successive unbranched edges are merged. Additionally, each read is associated with a path of edges, where the weight of that edge is determined by the number of reads mapping to the specific edge. The graph serves as basis for corrections of sequencing errors and contig generation.

It is worth mentioning that EULER-SR creates two k-mer graphs with different values for \( k \). As a result, it is possible to resolve longer repeats due to a larger \( k \), while gaining better connectivity in read areas with low coverage.

**Velvet – Zerbino et al. [ZB08]**

Zerbino et al. present Velvet, a set of algorithms used for de novo assembly. Plenty of other assemblers are influenced by the ideas of Velvet or reuse parts of it, as its algorithms have proven to be reliable and easy to use [MKS10]. Velvet does not exploit possible parallelism and does not focus on any multi-threaded execution.
Figure 2.7: Frequent de Bruijn graph structures, introduced by possibly faulty read data. (a) shows a tip, (b) an ambiguous link, (c) a bubble.

The created de Bruijn graph consists of nodes which are composed out of overlapping k-mers, an example can be seen in Figure 2.8b. Every node is associated with a *twin node*, representing the reverse series of the reverse complement. However, it does not have to be the reverse complement of its attached node. The twin nodes ensure that overlaps resulting from the opposite, potentially not sequenced, DNA strand are considered. Nodes are connected by *arcs* (i.e., edges), an arc is created if the outer k-mers of two nodes are the same.

To remove bubbles, an algorithm called *Tour Bus* is used. The resolution of repeats is performed by the *Breadcrumb* algorithm which leverages mate-pair information. In a later publication the authors present two other algorithms for repeat resolution and scaffold creation, called *Pebble* and *Rock Band* [ZMMB09], which can be applied in the Velvet tool-chain.

**ABySS – Simpson et al. [SWJ+09]**

Simpson et al. present ABySS, which was the first parallelized assembler using a distributed de Bruijn graph. K-mers represent nodes and are distributed across a cluster of computing nodes. It is necessary that the location of each k-mer can be calculated uniquely. The string representation of a k-mer is transformed to a base-4 value ($A = 0$, $C = 1$, $G = 2$, $T = 3$). A hash is calculated from this value and *xor*-ed with the hash value of the reverse complement. The location is then determined by the final hash value modulo the number of nodes in the cluster.

The authors do not state if the calculation of a k-mer’s location is done on
each computing node individually, or from a central point that sends the k-mers to their destination.

To store the graph’s adjacency information each graph node consists of the k-mer information itself and additional eight bits, where each bit marks the existence of an incoming or an outgoing edge. Figure 2.8c shows an illustration of the distributed nature of the graph used here. The adjacencies are determined after all k-mers are generated. For each k-mer an existence information is sent to computing nodes that might hold one of the eight possible neighbors. The distributed nodes check if they find an adjacent k-mer and possibly update their linkage information.

For the next steps, such as error correction and contig generation, the authors adapt algorithms introduced by EULER and Velvet. Due to the distributed nature of the de Bruijn graph communication between compute nodes is necessary, decelerating the algorithms compared to architectures using shared memory. However, the decreased time for graph creation outweighs the additional costs arising from communication.

ABySS is implemented using MPI which allows to increase the number of computing nodes as required. The authors do not inspect the behavior of their approach with varying numbers of computing nodes.

**ALLPATHS – MacCallum et al. [MPG+09]**

MacCallum et al. developed ALLPATHS, a de novo assembler for small genomes, later improved to cope with mammalian-sized genomes (ALLPATHS-LG [Gne11]). ALLPATHS performs an error-correction procedure on the read data set and identifies trusted k-mers. A k-mer is trusted if it occurs with a multiplicity above a selected threshold and of a certain quality. These k-mers are numbered and stored in a database. The same number is assigned to equivalent k-mers and overlapping k-mers get adjacent numbers, if possible. Two uniquely overlapping k-mers are merged to so-called unipaths, i.e., if two k-mers overlap in $k - 1$ bases, there must not exist a third k-mer overlapping in $k - 1$ bases, but with a distinct non-overlapping base. Hence, a unipath represents an unbranched sequence of bases in the original genome. See Figure 2.8f for an illustration.

The authors use the longest unipaths, called seeds. Seeds serve as starting points to assemble a subregion of the genome. By extending the seed in either direction a unipath graph emerges, which is similar to a de Bruijn graph. Thus, known error correction procedures, such as tip removal, can be performed. Compared to de Bruijn graphs, a unipath is an abbreviated, compressed path. The corresponding path in the de Bruijn graph consists of multiple nodes and edges.

ALLPATHS tries to preserve information on genomic ambiguity which might arise from unresolved repeats or SNPs and appends this information to its output.
2.4 Assembly Strategies

**SOAPdenovo – Li et al. [LZR+10]**

Li et al. developed SOAPdenovo with the intention to assemble human-sized genomes. An error correction procedure is applied to the read data set, relying on k-mer counts. According to the authors, this step is mandatory for large genomes. It reduces the overall number of distinct k-mers \((k = 25)\) of a read data set of the human genome from 14.6 billion to 5.0 billion. Li et al. use a de Bruijn graph representation similar to the one used by EULER and Velvet and found out by statistical analysis of the human genome that \(k = 25\) yields the best compromise. For further graph processing, the authors adopted algorithms from Velvet and enriched them to allow multi-threading. Two human genomes were assembled on a supercomputer with eight quad-core CPUs and 512 GB shared memory in 40 and 48 hours.

**Ray – Boisvert et al. [BLC10]**

Boisvert et al. implemented Ray, a de Bruijn graph-based de novo assembler that tries to leverage read data from different sequencing platforms, e.g., mixing Illumina data and Roche/454 data. The idea is to reduce the impact of sequencing errors, as different sequencing platforms produce errors with different characteristics.

The de Bruijn graph consists of k-mers of a multiplicity higher than a specifiable threshold as nodes. Edges are created between two nodes if the k-mers are overlapping within any read. Furthermore annotations are attached to the nodes allowing to infer the original read a k-mer originates from.

To generate contigs, Ray uses heuristics to extend selected sequences, so-called seeds. The heuristics make use of the annotated graph to decide how to extend a seed and into which direction. If no good decision can be made the process stops and a contig is emitted.

As an interesting fact, the authors remove reads that may be part of repeats. They found out that the quality of the resulting contigs is higher than if they were using all reads. Such reads are detected by finding subsequences of higher count than the coverage of the read data.

**PASHA – Lui et al. [LSM11]**

Li et al. present PASHA, an approach using a distributed de Bruijn graph that is created across several nodes in a computing cluster (strongly following the ideas of ABySS).

PASHA starts by distributing k-mers over all nodes of the compute cluster. Every node employs two processes \(T_0\) and \(T_1\), both of which has access to the whole read data set. \(T_1\) loads the reads from the hard drive and passes them in bundles to \(T_0\).

\(T_0\) extracts k-mers from the received reads and calculates a hash value for what the authors call a canonical k-mer. The canonical k-mer is the lexicographically
smaller of a k-mer and its reverse complement. Depending on the hash value and the number of overall processes, \( T_0 \) can decide whether to store the respective k-mer or to drop it. This assures that every k-mer is stored only once and in a deterministic fashion. PASHA only stores the canonical k-mer to reduce memory requirements, as either the original k-mer or its reverse complement can be inferred.

A first correction – the tip removal – can be applied on each node without any communication. Further correcting and simplifying algorithms may need information hold by other nodes, thus require inter-node communication. As a result, the time spent to create the graph itself is reduced, contrary to the time required to perform graph correcting algorithms.

As ABySS, PASHA is implemented using MPI. The authors examined how the approach scales in presence of different numbers of computing nodes. They found out that increasing the number of computation nodes decreases the overall execution time until a certain minimum is reached. Since only parts of the assembly are highly parallelized, this limit cannot be improved.

Conway and Bromage [CB11]

Conway and Bromage discuss succinct data structures to represent the de Bruijn graph using less memory. They use an approach leveraging entropy compression which reduces the memory requirements by a factor of 10. The entropy compression scales better in the presence of sequencing errors as well.

The authors do not store the k-mers, but \( \rho \)-mers, which represent the edges of the de Bruijn graph, e.g., if ACG and CGT are k-mers \((k = 3)\), ACGT is a \( \rho \)-mer. These \( \rho \)-mers are interpreted as integer values and used to address a bitmap of size \( 4^\rho \), where only one bit indicates the existence or non-existence of a \( \rho \)-mer, see Figure 2.8e for an illustration. To provide an efficient implementation of a sparse bitmap, the authors use a formalism introduced by Jacobs et al. [Jac89].

The k-mers are not stored explicitly, but are implicit due to the existence of a \( \rho \)-mer. To traverse the graph, a successor is determined by querying all possible four positions within the bitmap. Furthermore, Conway and Bromage optimize the way to store the counts of the edges since they claim that 32 bits per counter is more than necessary. Especially, erroneous reads may introduce plenty \( \rho \)-mers of very low count. Thus, they introduce a hierarchical representation, where by default only 8 bits per counter are used. If an edge occurs more often, they can extend these 8 bits to 16, or even 32 bits. The counters are stored separately from the bitmap, where a \( \rho \)-mer can query its frequencies in a constant lookup.

SparseAssembler – Ye et al. [YMC⁺12]

Ye et al. observed that memory requirements of state-of-the-art assemblers bound their execution to large and expensive super computers or compute clusters.
They propose to reduce the memory requirements by storing only a fraction of the k-mers stored by ordinary de Bruijn graph-based assemblers.

In their implementation, called SparseAssembler (successor SparseAssembler2), the authors build up a sparse k-mer graph. They only store roughly every $g$-th ($g < k$) k-mer and retain the skipped bases as additional linkage information. Hence, they increase the information stored in one node in the graph, while decreasing the overall number of nodes. Figure 2.8d illustrates such a graph node.

Ye et al. found that their approach reduces memory requirements by a factor of $1/g$. The sparse k-mer graph retains the graph traversal properties known from de Bruijn graphs, allowing error correction and graph simplification. E.g., bubble merging is performed by a Dijkstra-like breadth-first search, as introduced by Velvet. All of their algorithms run single-threaded and their costliest assembly of a human genome requires about one day runtime and only 29 GB of memory resources (as opposed to 40 hours and 140 GB for SOAPdenovo).

**SGA – Simpson and Durbin [SD12]**

Simpson and Durbin pursue the overlap consensus approach, and implemented an assembler called String Graph Assembler (SGA). They use the Burrows-Wheeler transformation in combination with a compressed FM-index to represent a read data set in a memory efficient way. The created FM-index serves as basis for the string graph construction and is queried during error correction and further graph processing.

An FM-index, introduced by Ferragina and Manzini [FM00], allows an efficient lookup within a compressed text. The Burrows-Wheeler transformation converts a textual input in a way, such that the output is optimized for compression, e.g., run-length encoding. A string graph differs from an overlap graph in removed duplicate reads, removed transitive edges, and some further optimizations. Thus, a string graph can be seen as a subgraph of the overlap graph.

The authors found that their implementation used 4.5 GB of memory for the assembly of a *C. elegans* genome (data set of 33.8 million paired-end reads). Compared to 23.0 GB, 14.1 GB, and 38.8 GB for Velvet, ABysS, and SOAPdenovo, respectively, this is a significant improvement. At the same time, performed quality evaluations of SGA showed competitive results. This also correlates with evaluations of Earl et al. [EBS+11], where SGA finished third. However, the execution time of SGA is still 3 times to 20 times slower than state-of-the-art de Bruijn graph assemblers.

Simpson and Durbin say SGA is the first assembler that is able to handle mammalian-sized genomes on low end computing clusters. They share the view of Miller et al. [MKS10], who assume that future sequencing technologies will increase the attractiveness of the overlap graph approach since the costly pair-wise overlap tests will reduce with increasing read lengths.
2 Basics of Bioinformatics

(a) K-mers of a de Bruijn graph can either represent the graph’s nodes or its edges.

(b) De Bruijn graph structure used in Velvet [ZB08]. Each node forms a compound with its twin node and can consist of multiple k-mers. Arcs connect nodes with each other.

(c) Distributed k-mers across computing nodes in a cluster, used by ABYSS [SWJ+09] and PASHA [LSM11].

(d) A node in the sparse k-mer graph employed by SparseAssembler [YMC+12]. Only every $g$-th k-mer results a node and branches can consist of up to $g$ bases.

(e) Conway and Bromage use a bitmap to mark the existence of an edge for their succinct data structures [CB11].

(f) Unipath graph as used by ALLPATHS [BMK+08]. K-mers are vertices, dotted ovals mark unipaths.

Figure 2.8: An overview of several de Bruijn graph representations in different de novo genome assemblers.
2.5 Assembler Comparison

Assemblathon 1 – Earl et al. [EBS+11]

Earl et al. describe Assemblathon 1, a detailed quality comparison of 41 assemblies. External groups performed the assembly of a simulated Illumina HiSeq short read data set, using a specific setup of assembly strategies and computing resources. De Bruijn graph-based assemblers were evaluated besides some OLC-based ones, such as SGA.

ALLPATHS and SOAPdenovo showed the best overall results, as next de Bruijn graph-based assembler ABySS placed 6th. The authors found that the best assemblies resulted in a high genome coverage and a good accuracy, but all of them have room for improvements. Thus, short read sequencing can compete with traditional Sanger-based technologies. According to Earl et al. earlier evaluations performed by MacCallum et al. [MPG+09, Gne11] showed the same conclusions.

Assemblathon 1 used only one single, simulated data set. To overcome this weakness and gain more insight into the quality of current assemblers, the authors propose to perform Assemblathon 2. The new study should include at least one mammalian-sized genome, and, as an important consequence, should analyze real data from different sequencing platforms as well.

Narzisi and Mishra [NM11]

Narzisi and Mishra performed evaluations of most of the state-of-the-art assemblers for short and long read data. Nearly all de novo assemblers, known at the time of evaluation, are represented.

The authors question if all current reference genomes are correct. And if not, how the assembly strategies need to be improved to provide correct reference genomes. They note that currently no standardized procedure for performance evaluation of assemblers exist. Furthermore, existing quality metrics, such as NG50, or contig length, are by far not sufficient enough. Sometimes they might even be misleading. For this reason they introduce a new metric called Feature-Response Curve (FRC). FRCs should provide a measurable quality assessment of the correlation between contig lengths and contig quality.

The performed evaluations use well-known metrics, e.g. NG50, but also include the new FRC metric. Narzisi and Mishra neither select a winner, nor do they create a ranking. They rather present the results of each metric in tabular form and oppose some of the weak ones with some of the good results. Overall, they found out that the quality of state-of-the-art assemblers varies dramatically.

An open-source software called FRCurve\(^1\) is available and the authors request scientists to produce exhaustive evaluations of assembly algorithms with this tool.

\(^1\)http://sourceforge.net/apps/mediawiki/amos/
Alkan et al. [ASE11]

Alkan et al. compare the results of a de novo short read genome assembly to a validated reference. They focus on the question if all genomic features are contained within a de novo assembly. It shows that assemblies drastically lack in completeness and miss many important features. Even though the authors justify this weakness with the short nature of the reads, they note that in their opinion scientists miss the importance of high quality assemblies and address this problem with too little effort. Thus, they highlight the requirement to apply a combination of reads produced by both high-quality and high-throughput sequencing technologies.

Salzberg et al. [SPZ+12]

Salzberg et al. performed an evaluation study of eight state-of-the-art short-read de novo genome assemblers called Genome Assembly Gold-Standard Evaluations (GAGE). They used four data sets of existing organisms, two bacteria, a bee, and the 14th chromosome of the human. Except for the bee, high quality reference genomes are available for detailed comparison. The authors found out that the most important thing to consider is data quality. The better the read data fed to any assembler the better the resulting assembly. Hence, they remark the importance of further improvements in sequencing technology itself.

Furthermore, the assemblies varied greatly in terms of quality and contiguity. While contiguity is a widely used metric for quality, Salzberg et al. worked out that some assemblies with superior contiguity contained several errors. This would be hard to detect without a comparison to an existing reference, thus reducing the overall quality. Besides, assemblers yielding good results for one genome might yield worse results for another genome.

According to the authors, ALLPATHS-LG shows the best overall performance. They conclude with the statement that their evaluation is only a temporary view of the state-of-technology as existing assemblers are under ongoing improvements.

Summary

Most state-of-the-art de novo genome assemblers applying any kind of a de Bruijn graph were discussed, see Table 2.1 for an overview. A broad variety exists in the specific representation of the graph, as well as in the strategies that are used to process the graph. The variety of different strategies and the frequency of new approaches, appearing every year, shows that great effort is being made to improve genome assembly in quality and execution time. On the same time it indicates that no perfectly fitting strategy has been found yet. A common finding is that de Bruijn graphs are a suitable representation in general but lack in two points. First, it requires extra effort to resolve repeats that are longer than a k-mer’s size. Second, the huge memory requirements for larger genomes bound the execution to costly computing hardware.
2.5 Assembler Comparison

<table>
<thead>
<tr>
<th>Name</th>
<th>Graph Representation</th>
<th>Genome Size</th>
<th>Year</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>EULER k-mer</td>
<td>nodes</td>
<td>small</td>
<td>2001</td>
<td>Pevzner et al. [PT01]</td>
</tr>
<tr>
<td>EULER-SR k-mer</td>
<td>nodes, merge paths</td>
<td>small</td>
<td>2008</td>
<td>Chaisson and Pevzner [CP08]</td>
</tr>
<tr>
<td>Velvet unipath</td>
<td>nodes, arcs</td>
<td>small</td>
<td>2008</td>
<td>Zerbino and Birney [ZB08]</td>
</tr>
<tr>
<td>ALLPATHS</td>
<td>graph</td>
<td>small</td>
<td>2008</td>
<td>Butler et al. [BMK+08]</td>
</tr>
<tr>
<td>ABySS</td>
<td>distributed k-mers</td>
<td>large</td>
<td>2009</td>
<td>Simpson et al. [SWJ+09]</td>
</tr>
<tr>
<td>Ray</td>
<td>graph represents read data</td>
<td>large</td>
<td>2010</td>
<td>Boisvert et al. [BLC10]</td>
</tr>
<tr>
<td>SOAPdenovo</td>
<td>shared memory</td>
<td>large</td>
<td>2010</td>
<td>Li et al. [LZR+10]</td>
</tr>
<tr>
<td>ALLPATHS-LG</td>
<td>unipath graph</td>
<td>large</td>
<td>2011</td>
<td>Guerre et al. [Gne11]</td>
</tr>
<tr>
<td>PASHA</td>
<td>distributed k-mers</td>
<td>large</td>
<td>2011</td>
<td>Liu et al. [LSM11]</td>
</tr>
<tr>
<td>Succinct DS</td>
<td>bitmaps</td>
<td>large</td>
<td>2011</td>
<td>Conway and Bromage [CB11]</td>
</tr>
<tr>
<td>SparseAssembler</td>
<td>g-sparsed k-mers</td>
<td>large</td>
<td>2012</td>
<td>Ye et al. [YMC+12]</td>
</tr>
<tr>
<td>SGA</td>
<td>overlap graph</td>
<td>large</td>
<td>2012</td>
<td>Simpson and Durbin [SD12]</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of the discussed de novo assemblers with focus on their interpretation and representation of a de Bruijn graph.

Simpson et al., Liu et al., and Li et al. try to overcome the memory-bottleneck by exploiting parallelism in existing strategies. However, this might shift the bottleneck to the communication which is required for graph processing. Ye et al.’s strategy to use only a subset of the overall k-mers targets modest size computers, even simple laptops.

With relation to an implementation on RIVYERA, a distributed de Bruijn graph fits perfectly since RIVYERA’s memory itself is distributed. Furthermore, RIVYERA’s communication can be highly optimized to the specific problem and the physical distances between the FPGAs are very small. Accordingly, the increased communication efforts are expected to perform better than those of existing approaches such as ABySS or PASHA. A closer description and assessment of the selected data structure is presented in Chapter 4 after the architecture of RIVYERA has been introduced in further detail in the next chapter.
CHAPTER 3

RIVYERA

RIVYERA\(^1\) (previously called COPACOBANA 5000) is a massively parallelized hardware platform that heavily relies on FPGAs. It is the successor of the COPACOBANA, which initially was developed to break the DES encryption algorithm using a brute-force attack. For further details on the attack see Kumar et al. [KPP\(^+\)06]. Besides cryptanalysis, other fields of application emerged among these are bioinformatics and the evaluation of investment strategies [WBB\(^+\)10, SGWS12].

RIVYERA's focal idea is to solve data and computation intensive problems using multiple FPGAs in parallel. An FPGA has the capability to execute highly optimized solutions for a certain problem, e. g., by exploiting problem inherent parallelism and constructing customized pipelines. RIVYERA connects over one hundred FPGAs using a bus system allowing communication between the computation units.

The machine is available in different versions, mainly diverging in the hardware equipment, e. g., the type and number of FPGAs. The concepts presented in this thesis target a RIVYERA S6-LX150. It consists of 128 FPGAs with two additional 256MB RAM modules per FPGA.

In the next section FPGAs are introduced shortly and their benefits are discussed. Furthermore, the hardware architecture as well as the API of the RIVYERA are explained in further detail.

3.1 FPGAs

FPGAs are integrated circuits that can be freely configured to fulfill a certain task. The development is done using a hardware description language such as VHDL or Verilog. An FPGA consists of hardware elements that can be combined or customized to perform a certain functionality (e. g., logic gates and look-up tables). Additionally, modern FPGAs contain further predestined units for special

\(^1\)http://sciengines.com
tasks (e.g., components for clock management, floating point computations, and RAM blocks). Leading manufacturers are Xilinx\textsuperscript{2} and Altera\textsuperscript{3}.

A Xilinx Spartan-6 LX150 consists of 11,519 Configurable Logic Blocks (CLBs). The CLBs contain two Slices, of which three different types with slightly different functionality exist. Each slice is comprised of four Look-Up Tables (LUTs), eight flip-flops and some additional logic. Furthermore, among others, 268 18kbit Block-RAMs and 6 Clock Management Tiles (CMTs) are available\textsuperscript{4}.

**Development**

As mentioned above, an FPGA’s design can be implemented in any hardware description language. As this thesis uses VHDL, a short example of the design flow of this language is given next.

VHDL strongly separates the *structural* description of a component from its *behavior*. The structural description is done by defining an *entity* with a *port* declaration. In the port declaration all incoming and outgoing signals are defined. Listing 3.1 shows an exemplary entity for a component eq\textsubscript{of\_two} which determines if two incoming 32 bit values are equal. In line 3 a clk signal is declared as input signal (in), line 4 and 5 show the 32 bit input values, and in line 6 the output (out) signal is declared, which is high if a\_in and b\_in are equal, low otherwise.

The actual behavior of the eq\textsubscript{of\_two} component is described in an *architecture*, see lines 11 to 24. The described *process* always waits for a rising edge of the clk signal and then either passes 1 or 0 to the output, depending on whether both inputs are equal or not. This behavior does not necessarily require a process that waits for a rising edge, but could also be composed of plain logic that updates the output as soon as one of the inputs change.

\textsuperscript{2}http://www.xilinx.com
\textsuperscript{3}http://www.altera.com
\textsuperscript{4}http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf
Listing 3.1: Example of a VHDL component that determines the maximal value of two input values at each positive clock edge.

This is just a very small example of a VHDL component. The language has many more features. For a detailed introduction you may refer to any introductory book on VHDL, e.g., Zwolinski [Zwo04].

Xilinx provides a fully-fledged IDE for the development process of their FPGAs, the ISE Design Suite\(^5\). After an FPGA’s design is written in VHDL, further steps are necessary until the FPGA can be programmed: Synthesis, Translate, Mapping, and Place and Route. During synthesis a so-called netlist is created. The written VHDL code is transformed into a logic circuit, i.e., gates (e.g., and-, xor-gates) and flip-flops. For further optimizations the target architecture is considered. The translation finds a corresponding circuit that is only made up of elements available to the specified FPGA. Mapping assigns these elements to the specific resources of the FPGA, e.g., logic functions are placed in LUTs. Finally, the required resources have to be placed in a physically available unit on the FPGA and the signals have to be connected.

3 RIVYERA

Figure 3.2: Schematical overview of RIVYERA’s architecture. Host system and FPGA computer are connected via a PCIe bus. The FPGA computer consists of multiple slot cards holding a ring of FPGAs. Figure adapted from [Rat12].

Benefits

A great benefit of FPGAs is the ability to be reconfigured with a totally different configuration. Accordingly, they are predestined for proof-of-concept implementations. Moreover, an FPGA’s design is customizable for the required task. A highly optimized and parallelized solution can be implemented and deep pipelines are possible. Though FPGAs are operated with a lower clock frequency than usual CPUs (e.g., 100 MHz opposed to 1GHz), a CPU has a bound instruction set and often requires multiple instructions. To perform the same task, an FPGA is possibly able to solve it in only one clock cycle. Furthermore, FPGAs often require less power and produce fewer heat than similar solutions [MC07, WBB+10].

3.2 Architecture

RIVYERA consists of a host PC and multiple extension cards, each comprising of several FPGAs, e.g., 16 extension cards, 8 FPGAs each. For applications that require memory, each FPGA has direct access to two 256 MB DDR-RAM modules. The extension cards are placed in a backplane where they are connected as a bi-directional array. Each card’s FPGAs are arranged and connected in a ring,
3.3 Communication API

Listing 3.2: A typical workflow of a host application controlling RIVYERA.

and the communication to other cards or the host are delegated by a service FPGA. The communication with the host is performed over a PCIe interface.

SciEngines provides an API for both the host and the FPGA allowing to implement the communication in a convenient fashion. Figure 3.2 shows a schematic overview of the architecture.

Host

The host PC is running a Linux operating system and is responsible for controlling the access to the machine and to initialize applications. An exemplary workflow is shown in Listing 3.2. The machine is allocated since multiple users can work on the host machine at the same time while the FPGA computer demands exclusive access. After the FPGAs are programmed with the design that should be executed, arbitrary communication between host and FPGAs can be performed. As soon as the desired task is finished, the machine is freed and available to other users.

3.3 Communication API

As mentioned above, FPGAs are able to communicate with each other, and the host is able to communicate with the FPGAs using the PCIe connection. Two scenarios can be differentiated. Either the host initiates a communication to any FPGA using the C/C++ API, or an FPGA communicates to a different FPGA (or the host) using the VHDL API. In the following, both APIs are introduced shortly.

Host

The C/C++ API provides two header files that specify the methods and types that can be used to interact with RIVYERA. The most important of these are listed below.

SE_ADDR: This struct is passed to all methods that address certain FPGAs. A slot, the specific fpga, and a certain register are specified. The register address can be used to distinguish different purposes of communication to the same FPGA. The slot and fpga addresses allow to
perform a broadcast to all available FPGAs or to all FPGAs of a certain slot.

**se_write:** This method allows to send an arbitrary amount of 64 bit words to a specified address.

**se_read:** This method reads a specified amount of 64 bit words from an FPGA. Three different read modes are supported. In active mode the host sends an explicit request (visible at the specified FPGA) and waits for as many data as requested to be send back. In passive mode the host just collects data that the specified FPGA has sent autonomously. The last mode (request) differs from the active mode in the sending of an explicit request, however, the host does not wait for any data to be returned. The upcoming data has to be collected using a passive read.

**se_waitForData:** This method provides functionality to wait until any FPGA of the specified address sends data.

**FPGA**

Every implemented FPGA design is wrapped by a top level component providing signals to interact with RIVYERA’s API. Basically, both the incoming and the outgoing communication are realized by a separate FIFO. As the signals for either direction are quite similar, only signals representing outgoing communication are listed below.

**api‌o‌_tgt‌._{slot,‌fpga,‌reg‌}.in:** The three signals are used to specify the recipient of the data. Special addresses are available for the host and for addressing all FPGAs.

**api‌o‌_tgt‌.cmd‌.in:** Specifies if a write is performed or if a read is initiated.

**api‌o‌.rfd‌.out:** The signal states whether the API is ready to process any communication. It can be seen as an inverted FIFO full signal.

**api‌o‌.data‌.in:** The signal holds the actual payload, one 64 bit word.

**api‌o‌.wr‌.en‌.in:** Pushes the current signals into the FIFO, and hence, initiates the communication for the current 64 bit word.
3.3 Communication API
CHAPTER 4

De Novo Genome Assembly on RIVYERA

The goal is to perform a de novo genome assembly on the massively parallelized hardware platform RIVYERA. A strategy for the representation and creation of a de Bruijn graph is presented. Algorithms to correct the graph and output contigs would use this representation to query and adapt nodes of the graph. Since they are not an integral part of this work and not yet implemented, they are not discussed in detail.

First, the overall concept is introduced, together with an optimized data structure to hold the de Bruijn graph. Later, the implementation of the FPGA design in VHDL, a reference implementation in C++, and the host application in C/C++ are presented and explained in detail.

4.1 Concept and Data Structure

For many algorithms the used data structure is the key to efficiency, convenient programming and understandability. Likewise, most of the genome assembly algorithms, presented in Section 2.4, use an adapted de Bruijn graph and only differ in the way they construct and use it. The de Bruijn graph is graphically descriptive, hence easily understood by developers. Large read datasets are condensed by storing overlaps as weighted paths in the graph, and efficient algorithms for graph processing can be implemented. Furthermore, existing de novo assemblers using de Bruijn graphs show promising results. To create a concept and a data structure fitting RIVYERA, the characteristics of the machine lead to certain requirements and benefits which are discussed in the following.

As mentioned in Section 2.1 a naive de Bruijn graph for mammalian-sized genomes requires hundreds of gigabytes of memory, which especially results from sequencing errors. Thus, a concise data structure is required. RIVYERA’s FPGAs themselves provide only a small amount of BRAM (XC6SLX150: 268·18 kbits = 603 kB), way to small to store a de Bruijn graph, even for small genomes. RIVYERA equips each FPGA with additional two 256 MB DDR-RAM modules, a total of 128·2·256 MB = 64 GB. The distributed nature of the available memory demands a distributed de Bruijn graph, such as used by ABYSS and PASHA. In
4 De Novo Genome Assembly on RIVYERA

a distributed de Bruijn graph each FPGA is responsible for only a fraction of the overall number of k-mers. Each k-mer’s location has to be computable quickly and deterministically.

The FPGAs access their DDR-RAM via a memory interface provided by Xilinx, which is programmed in VHDL. No high-level data structures are available per default, e.g., linked-lists or graphs. A data word can be configured to be of a size of 32 to 128 bits and is addressed using the memory’s row- and column-address.

For further graph processing, such as tip-removal or bubble merging, one FPGA might require information about a k-mer that is stored on a different FPGA. ABySS and PASHA use MPI to communicate with different computing nodes in a cluster and form larger packages to keep the number of communication events as small as possible. However, as MPI implementations provide an abstraction layer, the communication cannot be optimized to fit the specific task.

In general, a compute cluster has fewer nodes than RIVYERA’s 128 FPGAs. Accordingly, more communication is required on RIVYERA. Nevertheless, since the communication is very low level, directly between two FPGA using a 64 bit register, it can be highly optimized. Overhead can be reduced to a minimum and one FPGA’s communication does not interfere with another communication. The FPGAs communicate autonomously with each other over their own physical medium.

As mentioned before, the actual graph processing is not part of this thesis. Nevertheless, it is important to consider it during the development of a data structure, as the graph’s data structure has implications on the efficiency of potential correcting algorithms.

In short, following criteria were considered during the development of a graph data structure that fits RIVYERA.

**Conciseness:** Reduce memory requirements as much as possible.

**Efficiency:** Preserve efficient lookup of k-mers and allow efficient graph traversal.

**Realizability:** The structure has to be practical in VHDL.

**Concept**

Conceptually, the available memory, which is spread over the FPGAs, is seen as one address space. An illustration of this can be seen in Figure 4.1. A k-mer is converted to a binary representation, two bits for each base, and split into a prefix and suffix of certain length, see Figure 4.2 for a visualization. The prefix addresses a row of the memory, hence its length directly depends on the available amount of memory rows. Since the overall memory is split across the FPGAs, each FPGA stores only a certain number of prefixes. The ranges assigned to the different FPGA do not need to be of the same size. Section 4.1 discusses the reason for this. The suffix is stored together with a k-mer’s linkage information and a frequency counter as a 64 bit word within that row. The row itself is
4.1 Concept and Data Structure

Figure 4.1: The distributed memory is seen as one address space. Subsets belong to certain FPGAs. For instance, the rows starting from Row 1 belong to FPGA 1, the last rows until Row n belong to FPGA m.

arranged linearly, the first node to map into a row is positioned at the first spot, the second node (with different suffix) at the second spot and so on. The length of the suffix depends on the desired k and is bound by the available bits within the 64 bit word.

As already mentioned in Section 2.3, it is necessary to consider the reverse complement of each k-mer as well, since it is not known from which strand of the DNA the read origins. Storing both, the k-mer and its reverse complement, yields redundant information, as the existence of one sequence implicates the other one. Additionally, the occupied memory and the memory accesses would be doubled. For these reasons only the numerically smaller of a k-mer and its reverse complement, in binary representation, is stored.

Physically RIVYERA’s memory rows are of size 1 kB, hence each row can hold 128 nodes. Evaluations (see Section 5) showed that most rows hold far less than 128 nodes. Still, it is also possible that there exist more than 128 nodes with the same prefix. To overcome these problems, rows are not strictly arranged as they are physically. All requests of the graph creation performed here are random, i.e., it is not known if two adjacent requests address the same row. Therefore, no advantages of the physical row are lost. However, as the used rows are smaller than the physical row (size always a power of 2), all virtual rows are still located completely in a physical row. Thus, in the following, the term row does not necessarily refer to the physical row but of a row of arbitrary size. Secondly, a memory hierarchy is introduced to satisfy prefixes with an unusually large amount of different suffixes. Section 4.1 explains this in further detail.

The given RIVYERA setup has two memory modules per FPGA, one of which is used to store ordinary nodes, the other one is reserved for fallback rows. This yields an overall memory for nodes of \(128 \times 256 \text{ MB} = 32 \text{ GB} \), where as a first example rows are of the size of 512 B (64 nodes per row possible). Thus, the prefix used for addressing can be \(\log_2(32 \text{ GB}) - \log_2(512 \text{ B}) = 26 \) bits long, or in other words, 13 bases. Using a little trick, explained in Section 4.1, the prefix can be extended by one bit to 27 bits, or 13.5 bases.
Read: ACGTCAGATCATACA
K-mer: TCAGATC
Pre-/Suffix: [TCAG] [ATC]

Figure 4.2: Illustration of a k-mer’s storage. The extracted k-mer is split into prefix and suffix, the prefix is used to address a row of the memory, the suffix is stored together with a counter and linkage information in a cell of that row.

Node Representation

The de Bruijn graph’s nodes represent k-mers. Three different kinds of information have to be stored: The k-mer sequence, the linkage information extracted from reads, and the multiplicity of the k-mer (the number of occurrences of the specific k-mer within the overall set of reads). The linkage information is required to traverse the graph, the multiplicity of a k-mer is used to determine its quality.

In a naive graph data structure pointers to memory locations are used leading to the successor or predecessor nodes. Such pointers are unwanted as they require memory and possibly are superfluous, in case the location of a required node can be inferred.

The data structure used here is structured as follows. As mentioned earlier, a k-mer is split into prefix and suffix. The prefix of a k-mer is not stored explicitly, but is implicitly given by the row address at which the k-mer is located within the memory. Suffix, k-mer counter, and linkage information are stored in a 64 bit word. A k-mer has eight possible links, one for each base, either incoming or outgoing. The existence of such a link is stored by 1 bit, requiring exactly 8 bit for the linkage. The most significant bit of a word is used to indicate that this 64 bit word represents a valid node. The remaining 55 bits can be divided between suffix and k-mer counter.

Increasing the number of suffix bits allows bigger ks, but also increases the number of k-mers mapping to the same row, as suffix permutations increase. The k-mer counter should be large enough to allow a save distinction between k-mers of good and bad quality. In this thesis $k = 23$ is chosen. However, the $k$ as well as the size of rows, suffix, and prefix can be freely adapted to the needs. During evaluations (see Chapter 5) the values are changed, for instance, to gain insights into their impacts on execution times.

Optimizing Memory Structure

To store an additional bit implicitly, the aim is to rearrange the memory’s rows, such that one half of it will never be addressed. Hence, half of the memory is
4.1 Concept and Data Structure

Figure 4.3: Schematic overview of the performed memory rearrangement ($k = 3$). Of a k-pair only the smaller k-mer is stored. The bottom half of the memory is freed by mapping bottom k-pairs into free spots of the upper half. Free spots originate from upper k-pairs.

free and can either be used for additional information, or the original memory space can be chosen twice as large. Figure 4.3 illustrates this rearrangement procedure. For the sake of argumentation some terminology is introduced first.

**Definition 4.1 (Address Space).** The address space of a memory is split into an upper half, including the addresses $[0, \frac{\text{max address} - 1}{2}]$ and the remaining addresses, called bottom half. The maximal address of the lower half $\frac{\text{max address} - 1}{2}$ is also denoted by $H$, the maximal address by $M$.

**Definition 4.2 (revc and neg).** A bijective function transforming a k-mer $n$ (in binary representation) into its reverse complement $c$ is denoted by $c = \text{revc}(n)$. Likewise, a bijective function negating the bit representation of a number $n$ is denoted by $\text{neg}(n)$. Obviously, for any two numbers $a, b$ following equation holds: if $a = \text{revc}(b)$ then $\text{neg}(a) = \text{revc}(\text{neg}(b))$.

**Definition 4.3 (k-pair).** A k-mer along with its reverse complement is called k-pair. The smaller element (numerical in binary representation) of a k-pair $p$ is denoted as $p_S$, the greater element as $p_G$. If both elements of a k-pair are smaller than $H$, hence mapping into the bottom half of the memory, the k-pair is called a bottom k-pair, upper k-pair otherwise.

Considering that the memory is addressed using the prefix of the smaller element of a k-pair, those rows that would be addressed by the greater element of the k-pair are never used. At least if $k$ is odd such that no palindromes exist. See Figure 4.3 for an illustration. Obviously, this is true for exactly half of the memory rows.
To completely free the bottom half of the memory all bottom k-pairs are mapped into the upper half of the memory. This is done by subtracting the value of the smaller element of a k-pair from the maximal memory address $M$. Lemma 4.4 proves that for each bottom k-pair a free row in the upper half exists, if k is odd.

**Lemma 4.4.** For an odd $k$, each k-pair can uniquely be placed in the upper half of a memory.

**Proof.** For each k-pair with at least one of its elements mapping into the upper half of the memory the assumption is obviously true by the definition of the k-pair. Thus, following the assumption is shown for k-pairs with both elements mapping into the bottom half.

Let $p$ be an arbitrary bottom k-pair, in other words, both elements of $p$ map into the bottom half of the memory. Now one has:

\[
\mathcal{H} < p_S < p_G \\
\iff \mathcal{H} \geq \neg(p_S) > \neg(p_G)
\]

Since $p_S = \text{revc}(p_G)$ (definition of the k-pair) it is also true, that $\neg(p_S) = \text{revc}(\neg(p_G))$. Thus, for any $p_S$ there exists a spot in the upper half of the memory ($\neg(p_S)$) with a smaller reverse complement ($\neg(p_G)$, as $k$ is odd $\neg(p_S) \neq \neg(p_G)$ is true), such that the corresponding upper k-pair is stored at $\neg(p_G)$ and $\neg(p_S)$ remains free – $p$ can be stored at $\neg(p_S)$. Due to the fact, that revc and neg are both bijective, it is not possible that two different k-mers map to the same spot. 

### Addressing

The node representing a k-mer is stored in a row of the memory that is directly addressed using the k-mer’s prefix. Together with Lemma 4.4 it is possible to define a function to compute the address for k-pairs, where all addresses map to the upper half of a memory.

**Definition 4.5** (kaddr). The function $kaddr$, which computes the address of a k-pair $p$, is defined as follows:

\[
kaddr(p) = \begin{cases} 
    p_S, & \text{if } p \text{ upper k-pair} \\
    \mathcal{M} - p_S, & \text{else}
\end{cases}
\]

FPGAs are only responsible for a defined interval of the memory. This interval is specified by two integer values which directly represent the first prefix and the last prefix handled by that specific FPGA. To calculate the address from the view of an FPGA another function is defined as follows:
4.1 Concept and Data Structure

Figure 4.4: Overview of the memory hierarchy used to store k-mer nodes. If a prefix-addressed row is full, the suffix memory will be tried using the k-mer’s suffix. If the suffix memory’s row is also filled, the k-mer will be stored in the tree memory.

**Definition 4.6** $(kaddr_i)$. The function $kaddr_i$ calculates the address of a k-pair $p$ for an FPGA $i$, where $ranges$ represents an array holding the first prefix of each FPGA, and is defined as follows:

$$kaddr_i(p) = kaddr(p) - ranges[i]$$

**Memory Hierarchy**

Evaluations showed that a memory with a fixed row length results in several overflowing rows. Due to the heterogeneous data, some rows need to hold far more distinct k-mers than the row size. Thus, it is necessary to provide space that holds these k-mers. This is done by splitting the overall available memory into three different types. The second type is addressed in the same fashion as before, but this time the suffix of a k-mer is used for addressing instead of its prefix. The memory addressed by prefixes is called **prefix memory**, the one addressed by suffixes is called **suffix memory**.

The idea behind this is the observation that some of the overflowing rows do it heavily, i.e., multiple times of the actual size. For such a row, all k-mers that are assigned to that specific row are equal in prefix, but differ in suffix. Hence, all k-mers that overflow the prefix memory, will address a different row in the suffix memory. Figure 4.4 shows a schematic overview of this procedure.

As it is still possible that also the rows in the suffix memory overflow, a third type of memory is introduced. Using prefixes to implicitly address rows, allows to find a k-mer’s spot very fast but leaves a sparse memory with many unused, still reserved, rows. Furthermore, there is no concept to enlarge a row such that it can hold as much k-mers as required. To provide a memory structure that yields a dense memory utilization and still has reasonable query times, a binary search tree is introduced.
For the tree memory, a single node has to store the whole k-mer (not only the suffix of prefix), as no implicit addressing is possible anymore, and two pointers to possible successors. Hence, instead of 64 bit each node now requires 128 bit. The nodes can be created on demand and be placed in the next free cell of the memory. Inserting and querying a certain k-mer has averagely a logarithmic complexity.

**Load Balancing**

Due to the diverse nature of genomic read data the work load of the distributed FPGAs might deviate strongly. With equidistant prefix ranges the work load is very imbalanced, see Figure 5.1 in the evaluations section. Which and how many k-mers an FPGA handles depends on the prefix of that specific k-mer and which prefix range is assigned to the FPGA. Thus, knowing the number of occurring prefixes beforehand can help to assign non-equidistant prefix ranges to the FPGAs. Larger ranges will be assigned if fewer k-mers with a matching prefix are present, smaller ranges if many k-mers with a certain prefix will fall into that range. Depending on the size of the prefix and the available memory the range assigned to an FPGA is bound to a maximal size, smaller ranges leave unused memory rows.

To calculate non-equidistant ranges, prefixes are counted prior to the actual graph creation. As prefixes are in the order of 27 bits, it is still possible to use a counter for each possible prefix requiring only 1 GB of the memory. Using the result of these counters the non-equidistant ranges can be determined. If the prefix is too large, it is possible to reduce the counter size and count multiple prefixes with one counter. This introduces some inaccuracy but still yields better distributions.

For the sake of argumentation, the described problem is formalized as a scheduling problem, where FPGAs are machines, the prefix counters are denoted as indexed jobs, and the assignment of possibly multiple jobs to the machines is referred to as schedule. The jobs assigned to the same machine must form a contiguous interval, the width of this interval is bound by the available memory per FPGA and is denoted as \( M_{int} \).

**Definition 4.7 (Job).** Let \( J \) denote a set of jobs. The jobs are enumerated as \( j_x \), where \( x \in \{0, 1, ..., |J| - 1\} \). Each job \( j \in J \) has a weight, e.g., representing the effort to fulfill the job as compared to the other jobs, written as \( j.weight \).

**Definition 4.8 (Job Interval).** A job interval is a contiguous sequence of jobs denoted as \( j_s...j_t \), where \( s, t \in \mathbb{N}_0 \) and \( s < t < |J| \). Let \( i = j_s...j_t \) be a job interval, functions **weight** and **size** are defined as follows:

\[
\begin{align*}
i\.weight &= \sum_{r=s}^{t} j_r\.weight \\
i\.size &= t - s
\end{align*}
\]
Definition 4.9 (Schedule). A schedule $S$ is a sequence of job intervals. The sequence must contain as many elements as machines are available.

Definition 4.10 (Valid Schedules). Let $m$ denote the number of available machines and let $J$ be a set of indexed jobs to be scheduled. $\mathcal{M}_{\text{int}}$ denotes the maximal job interval size. The set of valid schedules $\mathcal{I}_V$ is defined as follows, where $\mathcal{I}$ holds all possible schedules with ordered job intervals.

\[
\mathcal{I} = \{(j_{s_1}, j_{e_1}, j_{s_2}, j_{e_2}, \ldots, j_{s_m}, j_{e_m}) : 0 = s_1 < e_1 < \ldots < s_m < e_m = |J| - 1\}
\]

\[
\mathcal{I}_V = \{S \in \mathcal{I} : \forall i \in S : i.\text{size} \leq \mathcal{M}_{\text{int}}\}
\]

Definition 4.11 (Bound Schedule). A schedule $S$ is said to be bound by a value $x \in \mathbb{R}$, denoted as $\bar{S}$, if following holds:

\[
\forall i \in S : i.\text{weight} \leq x \text{ and } \forall y \in \mathbb{R} : y < x : \exists i' \in S : i'.\text{weight} > y
\]

Definition 4.12 (Optimal Schedule). Within a set of valid schedules $\mathcal{I}_V$, the optimal schedule is the schedule with minimal weight on all intervals and is defined as follows:

\[
\hat{S}_{\text{opt}} = S \in \mathcal{I}_V : \forall S' \in \mathcal{I}_V : \bar{S} \leq \bar{S'}
\]

$\hat{S}_{\text{opt}}$ is also referred to as optimal weight $\mathcal{M}_{\text{w}}$. $S_{\text{opt}}$ does not necessarily exist and might be ambiguous.

Lemma 4.13. Let $J$ be an arbitrary set of jobs and $m$ the number of machines. For any schedule $S$ the bound $\bar{S}$ is not smaller than $L = \frac{\sum_{j \in J} j.\text{weight}}{m}$.

Proof. Assume that there exists an $L'$ with $L' < L$ such that $S$ is bound by $L'$. Now it is true that $\forall i \in S : i.\text{weight} \leq L'$. As a consequence the following holds:

\[
\sum_{i \in S} i.\text{weight} \leq m * L' < \sum_{j \in J} j.\text{weight}
\]

However, this is a contradiction since, by definition, the sum of interval weights cannot be smaller than the weight of all jobs.

Following, an approximation algorithm to calculate a near-optimal schedule (prefix assignment to the FPGAs) is presented.

The Algorithm  Listing 4.1 shows the pseudo code of the approximation algorithm. The target is to find a distribution of the available jobs among all machines (FPGAs), such that each machine gets assigned about the same weight. Line 1 to 5 introduce some basic variables. The jobs variable is an array holding all available jobs. In the intervals array the assignment of job intervals to machines is stored, hence the array’s size equals the number of machines. The maximal interval size is hold by mint, the desired weight for each machine
Listing 4.1: Load-balancing algorithm used to balance the number of handled k-mers per FPGA as good as possible.

4 De Novo Genome Assembly on RIVYERA

by $mw$. Initially, the theoretically optimal value of the overall weight divided by the number of machines is chosen for $mw$ ($mw$ is adapted if no schedule can be found). Line 7 and 8 hold variables that indicate the job interval ($jint$) currently worked on and a flag whether the latest schedule is valid or not.

The algorithm is iterated as long as a valid schedule is found (see line 10). Each time all jobs are considered and it is tried to partition them into job intervals (see line 11 to 22). Each job interval starts empty and gets jobs appended until either the maximal weight ($mw$) or the maximal interval size ($mint$) would be exceeded, see line 14. If neither value is exceeded, the current job’s weight is added to the current job interval’s weight and the size will be incremented (see line 20 to 21). If a job interval cannot take more jobs, the next job interval will be started (line 16).

The job loop finishes if all jobs are distributed or it is interrupted if the last job interval cannot take all remaining jobs, resulting in an invalid schedule (see
4.1 Concept and Data Structure

In case of an invalid schedule the desired bound is adapted. As adaption any increased value is valid. Here, the weight of left over jobs divided by the number of machines that did not hit the maximal interval size yet is added to the previous value of \( mw \).

In the following, it is shown that the approximation algorithm finds a near-optimal schedule.

**Lemma 4.14.** Let \( L \) be the desired bound of an arbitrary round of the algorithm. If the algorithm does not find a valid schedule \( S \), such that \( \bar{S} \leq L \), there does not exist a schedule \( S' \), such that \( \bar{S}' \leq L \).

**Proof.** Only the case in which the algorithm does not find a valid schedule is considered here. The algorithm starts at job zero and combines job intervals out of jobs as long as the two requirements (weight less than \( L \) and size smaller than \( M_{int} \)) are met. Thus, every finished job interval cannot be enlarged with a further job.

At the end of the while loop (line 23) not all jobs were processed (otherwise a valid schedule is found). Hence, if a schedule \( S' \) would exists, the existing job intervals can be rearranged such that the remainder of the jobs can be distributed as well. For each job interval there are two possibilities: to pass jobs to the left (preceeding), or to pass jobs to the right (succeeding) job interval. If it was possible to pass a job to the left, the algorithm would have done this in the first place. Passing a job to the right reduces the weight of that specific job interval but increases the weight of the job intervals on the right. This yields either no change in the remaining jobs or in an increased number of jobs that are left over.

Thus, if the algorithm does not find a schedule bound by \( L \), no such schedule exists. \( \square \)

**Lemma 4.15.** The algorithm finds a schedule \( S \) such that \( S_{opt} \in [\bar{S} - \delta, \bar{S}] \), where \( \delta \) indicates the increase of the desired bound before the last iteration.

**Proof.** The algorithm starts to try to find a schedule with a bound of \( \sum_{j \in J} j.\text{weight} \). According to Lemma 4.13 this is a valid starting point as no lower bound can exist. With Lemma 4.14 it can be seen that there cannot exist a valid schedule with a bound lower than \( \bar{S} - \delta \) (which represents the second to the last result). As the algorithm always finds a schedule (in the worst case the bound equals the total weight of all jobs), it is obvious that \( S_{opt} \) has to be in the stated interval. \( \square \)

**Runtime** There are two loops to be considered. The inner loop, which runs through all jobs and the outer loop which runs until a valid schedule is found. The runtime of the outer loop depends on the input data and the implementation of the adapt function. For instance, if the input data is perfectly schedulable, i.e., no job interval hits the maximal interval size, the outer loop is executed exactly once. Likewise, if the adapt function increases \( mw \) in very large units, it
will take less steps than with a very sparsely increasing function. However, the result might be closer to the optimum when adapting in small steps.

In any scenario, the inner loop runs through all jobs or until it might be interrupted. Therefore, the algorithm has a quadratic runtime in the worst case. However, evaluations indicate that for the used adaption function the outer loop does not require an intolerable number of iterations (less than eight iterations for all used read data sets, see Table 5.3).

Moreover, in a complete assembly pipeline a stage would be included that performs a pre-correction of the read data. Such a pre-correction often counts k-mers and makes decisions based on these counts. Thus, every read is already looked at prior to the graph creation – the perfect place to incorporate the counting of prefixes necessary for load-balancing. Executed on an FPGA the pre-correction does not need more cycles since the counting can be realized in parallel to existing work. Furthermore, even the algorithm described above can be run in parallel to any processing of the pre-correction. Hence, the load-balancing algorithm can be run without introducing any additional execution time.

**K-mer Queries and Graph Traversal**

Graph processing algorithms used to detect and repair erroneous paths within the de Bruijn graph require to query for adjacent nodes in either direction. For a given k-mer \( k \), and a successor \( k' \), with equal \( k-1 \) bases, one has to perform the following calculation to determine the address at which \( k' \) is stored. First, the required k-mer has to be mapped according to the rules introduced in Section 4.1 (select the smaller of the k-mer and its reverse complement and possibly map it into the upper half of the memory). From this value the prefix can be extracted, which the FPGA either owns itself or which has to be queried from a different FPGA. With the introduction of non-equidistant prefix ranges the location of the prefix cannot be calculated statically anymore. Thus, either every FPGA must keep record of the current ranges of all FPGAs or a request has to be sent to all FPGAs until the correct one is found. In any way, the communication between the FPGAs has to be realized in an efficient way. Assemblers such as PASHA and ABySS use a distributed graph and indicate that during steps such as bubble-merging the time required for communication has the greatest impact on the execution time of the contig generation [LSM11, SWJ+09].

**Communication** RIVYERA allows very direct and low-level communication of 64 bit words between FPGAs. However, every FPGA is only connected to its neighbors on the same card. Likewise, every cards is only connected to its neighbors respectively. For this reason, random communication between any two FPGAs would yield lots of messy data paths across the internal bus. A directed data path optimally leveraging RIVYERA’s architecture is favorable.

A possible solution works as follows. Each FPGA employs a small control unit, which is able to receive a data package in every cycle and either accepts the
package for internal processing or just forwards the package to a neighbor. This way, the bus can be fully saturated as every FPGA’s communication only targets the direct neighbor. A concise protocol describing requests and responses can be developed. A query for a certain k-mer is packaged as a request and placed on the bus. There it passes through all uninterested FPGAs and is finally picked up by the FPGA that holds (or at least might hold) the required k-mer. The responsible FPGA packages a response with the requested data and the requesting FPGA as the target and places the package on the bus.

**K-mer Lookup**  The finding of a k-mer within the memory works in the same way as the storage. First, the prefix of the k-mer is used to address a row within the prefix memory. If the corresponding suffix cannot be located, the suffix memory will be checked. If this fails as well, the k-mer will be searched for within the tree memory. If a leaf (not equaling the request) is reached within the tree memory, the required k-mer is not available.

**Deletion of Nodes**  During tip removal or bubble merging the deletion of certain k-mer nodes is necessary. A deletion of a node in the prefix memory would imply to copy all subsequent nodes to an earlier spot within the row (similar for the tree memory, where the deleted node has to be replaced by either the ordered predecessor or successor). Alternatively, a deleted node can be marked with a flag and just be ignored the next time the row is traversed for a different k-mer. Which of the two approaches is costlier depends on the amount of graph traversals after nodes have been deleted. If there are a lot of queries afterwards and the deleted nodes are passed very often, it might be less costly to rearrange the memory rows.

This theoretical approach allows arbitrary graph traversal and communication between any two FPGAs. Its implementation and evaluation is not part of this thesis but indicates that further steps of the assembly pipeline can be realized on RIVYERA.

**Alternative Data Structures**

The presented de Bruijn data structure is optimized for RIVYERA’s architecture and the direct, low-level access of the available RAM. In the following, data structures that were discarded during development are discussed shortly, especially with relation to reasons for their unsuitability.

**Graph**  Alternatively to the presented data structure an ordinary graph, using pointers, could be implemented. To realize such a structure in VHDL a node must hold additional pointers to its successor nodes. A de Bruijn graph has one successor node for each of the four bases. If only outgoing connections are stored, four pointers are required, otherwise, storing both directions, additional eight pointers are required. Thus, each node has to be enlarged by \(\{4, 8\} \cdot m\).
De Novo Genome Assembly on RIVYERA

bits (where \( m \) is the address width of the memory, e.g., 30). The space for each pointer has to be kept free, even if a successor does not exist (rearranging the memory would be costly). If a node of the de Bruijn graph results from an error free read, three pointers in either direction will be empty, wasting memory.

As a result, a graph connected by pointers requires at least doubled memory. The fact, that in a de Bruijn graph it is inferable for each node how the k-mer of an adjacent node looks like, makes the pointers superfluous. Each of the four pointers can be inferred unambiguously, as a node’s location can be determined using the k-mer sequence as the key. Thus, removing the pointers has no impacts on the traversability of the graph.

Hash Map Using a calculated hash to address a memory cell implicates that the hash is equal, or smaller, in length to the memory’s address. For instance, with the present setup of the RIVYERA, this is 26 bit. Hence, for \( k = 23 \) the hash function has to reduce 46 bit to 26 bit, which most likely results in collisions. To retain full information about the graph, and allow graph traversal, these collisions need to be resolved in a deterministic fashion, i.e., mapping back to the original k-mer. Thus, for each collision additional information about the originating k-mer has to be stored, resulting in further memory requirements and complexity.

Sparse Bit Map Conway et al. use a sparse bitmap to store the existence of an edge (they call it \( \rho \)-mer) within the graph. They emphasize, that for \( \rho = 26 \) such a bitmap requires 512TB of memory. Hence, they use high-level implementations of a sparse bitmap, which stores the sparse information in a very compressed way. \( \rho - \text{mer} \) counts are stored in separate vectors, which again make use of compressed data structures. At first sight, a VHDL implementation of such data structures appears to be rather complex and time-consuming. It might make sense to evaluate sparse bitmaps in the future.

4.2 Implementation

In this section the actual implementation in C/C++ and VHDL is explained. First, a general overview of the components as well as the flow of information are illustrated, see Figure 4.5 for a schematic overview. Second, the control application (host) is presented, especially the interaction between host and FPGAs. Third, the design running on each FPGA is discussed in detail and a reference implementation developed in C++, imitating the FPGA design, is shown. Finally, in Section 4.2, methods to validate the results of the implementation are discussed.
4.2 Implementation

Figure 4.5: A schematic overview of the components used to perform the creation of a de Bruijn graph using a distributed memory architecture.

Overview

The left hand side of Figure 4.5 shows the controlling host unit. It has access to the genomic read data from which the de Bruijn graph should be constructed. Each read is broadcasted to all FPGAs, where they are first processed by a unit called K-mer Generation. This unit divides a read of length \( r \) into \( r - k + 1 \) k-mers (Read Windowing), processes each read as described in Section 4.1 to reduce memory consumption (K-mer mapping), and decides whether the k-mer belongs to the current FPGA. Afterwards, the k-mer will be passed to a unit that accesses the Prefix Memory. If the k-mer’s row is already filled, the k-mer is passed to a unit controlling the Suffix Memory. In case the suffix memory’s row is also full, the k-mer will be passed to a unit that places the k-mer in a binary search tree. As illustrated in Figure 4.5 the components are arranged as a pipeline.

Evaluations showed that the prefix memory has to cope with the most accesses (see Figure 5.3), hence it has its own physically separated RAM module. The suffix memory shares the module with the search tree, however, accessing distinct parts of the memory. The physical separation allows a parallel processing of the insertion into either prefix memory, or suffix and search tree memory as well.

Control Application

The host application is implemented in C/C++ and makes use of the API introduced in Section 3.3. Its task is to load the read data from disk and pass the reads to every FPGA of the RIVYERA.

The API uses 64 bit words for communication. Thus, reads are transformed into a binary representation. Again, 2 bits are used to encode a base. Furthermore,
4 De Novo Genome Assembly on RIVYERA

1 var buffer; // infinite buffer
2 var readData;
3 var done = false;
4
5 T1:
6 for read in readData {
7     buffer += read.length;
8     buffer += convertRead(read);
9 }
10 done = true;
11
12 T2:
13 var lastPos = 0;
14 while !done || (lastPos < buffer.size) {
15     if buffer.size - lastPos > batchSize {
16         broadcast(buffer.getBatch(lastPos));
17         lastPos += batchSize;
18     }
19 }

Listing 4.2: The two threads of the host program. T1 is loading reads, T2 sends them to the FPGAs.

reads can be of different length which requires the read’s length to be sent as well. The current implementation sends five 64 bit words per read, the first word represents the read’s length, the latter four words represent the read in binary representation split into 64 bit words. Hence, a read currently cannot exceed a length of 128 bases. Increasing the maximal read length, or sending just as many 64 bit words as necessary is possible.

In a nutshell, the host application fulfills the following steps, of which steps 3 and 4 are discussed in further detail.

1. Allocate the machine and program the available FPGAs.
2. Inform each FPGA about the prefix range it is responsible for.
3. Load the reads from disk, convert them into the binary representation explained above, and buffer each 64 bit word.
4. Broadcast the converted 64 bit words to all FPGAs.
5. Wait until all FPGAs have finished.
6. Allow to run optional evaluations, e.g., count k-mers on each FPGA (only debugging and evaluation).
7. Free the machine.

Listing 4.2 shows pseudo code performing the functionality for steps 3 and 4. At this, thread T1 represents step 3, see lines 5 to 9. Reads are loaded from
their storage, e.g., the hard drive, converted into their binary representation, and stored to a buffer together with their length. T2 takes the elements out of the buffer and broadcasts them in larger batches (to allow fast communication) to the FPGAs (lines 12 to 19).

**FPGA Design**

In this section the core elements of the implementation in VHDL are presented in further detail. However, the data flow between the components as seen in Figure 4.5 is discussed, whereby the program code is omitted. Accordingly, each component’s incoming and outgoing data signals are illustrated.

All components form a pipeline and each component acts autonomously, i.e., they do not require global control signals that direct their behavior. As soon as data arrives in a component’s input FIFO, the component performs its task and possibly passes the created data to the next FIFO. The data flow is unidirectional, as this implementation only builds and stores the graph.

**Top Level Component**

![Top Level Component Diagram]

**Description:** The top level component creates all subcomponents and is responsible for two main tasks. First, it processes the communication with the host. Reads split into 64 bit words are received (api_i_data) and assembled to one 256 bit vector. Together with the current read’s length this vector is passed to the internal component generating k-mers. Second, it delegates all interaction with the RAM modules (mem_port*).

Furthermore, before the host sends reads, this component is responsible for receiving the values of the first handled prefix and the last handled prefix. These two values are passed to subcomponents that require this information.
K-mer Generation

Description: As seen in Figure 4.5, the k-mer generation is split into two parts. The front component splits the incoming read (in_read_din) into its k-mers depending on the specified $K$ and the current read length (in_readn_din). Likewise, linkage information is extracted, i.e., the previous base in the read and the next base in the read (in case they exist). The extracted k-mers are placed in an internal FIFO and processed by the second component. First, the k-mer is transformed according to the mapping rules introduced in Section 4.1 (pick the smaller of a k-mer and its reverse complement and possibly map it to the upper half of the memory). Second, it is checked whether the k-mer is handled by the current FPGA, i.e., if its value is smaller or equal to in_first_prefix and greater or equal to in_last_prefix. If this condition holds, the transformed k-mer is passed along with its linkage information to an output FIFO (out_data), otherwise it is discarded.

Memory Access

Description: The interface of all three memory accessing components (prefix, suffix, and tree memory) is equivalent. Node data produced by the node generator is passed to the component (in_data). The prefix memory uses the in_address_offset signal to determine the actual row location within the memory. The suffix and the tree memory do not require this
4.2 Implementation

information, as they directly use the suffix as memory location or place the node within the binary tree, respectively. In case the end of a row is reached, the data is passed to the next memory unit (out_data), as seen in Figure 4.5. The requests to the memory are performed using the signals in_ram_port and out_ram_port. The specific configuration (e.g., prefix or suffix memory row widths) is done using VHDL generics.

**Node Insertion**

The internal process of the memory accessing units is implemented as a state machine and illustrated in Figure 4.6. Upon an incoming k-mer its prefix (depending on the memory type) is extracted and used to address and load the first element of a row. If this is an existing node, the suffixes are compared. In case the suffixes are equal, the linkage information and k-mer counter is updated and the node is stored to memory. The next node of the addressed row is loaded if the suffixes are different and the row holds further nodes. Reaching the end of a row results in a redirection of the k-mer to the next memory in the memory hierarchy.

Provided that no node has been placed at a loaded spot in the memory row, a valid bit is set, the suffix is stored, linkage and counter are set initially, and the node is stored to memory. Afterwards, the state machine transitions into its idle state and waits for a new incoming k-mer.

The component representing the tree memory behaves in a similar fashion. However, instead of loading the next element within a memory row, a previously stored pointer to a memory cell (as part of the node) is used to load the next node. Accordingly, upon creation of a new node, this pointer is stored along with the whole k-mer (no bits are stored implicitly as row address), the linkage information, and a counter.

**Resource Utilization**

As expected, the Spartan6-LX150 provides far more resources than required, see Table 4.1 for the exact figures. Opposed to the required memory to hold the de Bruijn graph, the logic used to perform the graph creation is rather small. The presented figures already include RIVYERA’s API. The highest resource utilization is found for the 16 kbit BRAMs (44%). All other resources are utilized below 10%. Hence, there is enough space on the chip to perform other algorithms, such as an preliminary error correction or subsequent graph processing algorithms. In case further block RAM is required, the 44% can be decreased. The graph creation uses block RAM for FIFOs, whose depth can be reduced without having impacts on the saturation of the pipeline.
4 De Novo Genome Assembly on RIVYERA

Figure 4.6: Illustration of the state machine used to insert k-mers into the distributed de Bruijn graph, as used by the VHDL implementation of the prefix memory and suffix memory.

**Reference Design**

During the span of this thesis no fully equipped RIVYERA was available. However, to provide exhaustive evaluations a reference implementation in C++ was created. This implementation behaves as closely to the FPGA design as possible. A single FPGA is modeled as a thread, receiving 64 bit words via a blocking queue. The run method of the thread implements the FPGA’s functionality. For simplicity, the subcomponents discussed in the previous section are realized as methods. As an example, the creation of k-mers is done by calling a method createKmers with the read as input and a pointer to a buffer. After the method returns, the buffer contains the k-mers that are valid for the specific FPGA.

To obey memory limits on a standard computer (probably less than $128 \times 512$ MB = 64 GB of RAM) the implementation of the memory has to be done in an efficient way. The notion of memory rows is retained, however, as the backing data structure Google’s sparsetable\(^1\) is used. A sparsetable is similar to an array.

\(^1\)http://code.google.com/p/sparsehash/
### 4.2 Implementation

<table>
<thead>
<tr>
<th>Slice Registers</th>
<th>Slice LUTs</th>
<th>Occupied Slices</th>
<th>BRAM16</th>
<th>BRAM8</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,287 of 184,304</td>
<td>5,163 of 92,152</td>
<td>1,801 of 23,038</td>
<td>118 of 268</td>
<td>12 of 536</td>
</tr>
<tr>
<td>2%</td>
<td>5%</td>
<td>7%</td>
<td>44%</td>
<td>2%</td>
</tr>
</tbody>
</table>

Table 4.1: Resource utilization of the current FPGA design.

...in that it has a certain size and provides `get` and `set` methods for its cells. It trades memory efficiency with execution time, requiring only 1 bit per unused value. Due to the fact that the graph creation leaves most parts of the memory free, using the sparse table saves a lot of memory. Accordingly, the prefix and the suffix memory are implemented as sparsertables (rows) in a sparsetable, where the columns are created on demand, i.e., only if a k-mer is to be placed in that row.

Implementing the binary search tree is straightforward. A `struct` holding a k-mer's information and two pointers to possible successors is used. New nodes are created only when necessary.

Obviously, the simulated FPGA is a sequential program with a drastically higher runtime compared to the actual FPGA design. However, the execution time of the C++ program is of no interest (performance is inferred using the measured data), as long as it finishes in an acceptable time.

It is desirable to use the previously created host program for the execution of the simulated FPGAs. As an abstraction layer, a header file matching the exact interface of the RIVYERA API is implemented. This way it is possible to use the unaltered host program, introduced in Section 4.2, to run the reference implementation.

### Validation

The distributed de Bruijn graph created in this work is unique in its structure. Moreover, no frameworks are available that provide standardized validation methods.

Validating the FPGA design is performed in two steps. First, all subcomponents are tested using testbenches. A testbench can be seen as a unit test of a VHDL file and contains small examples of input data for the subcomponent. Validation needs to be done manually in ISE’s simulator by checking the corresponding results.

To validate the overall functionality of the FPGA design and the host application, an additional component is added to the VHDL design. A `graph-sender` component runs through the memory and sends every node it finds to the host. The host converts the binary representation to a textual output format as discussed shortly.

Second, the reference implementation is used, since it emulates the structure and proceeding of the overall VHDL design. The de Bruijn graph is built internally...
from a set of read data and printed out in a text format, one line per k-
mer node – the k-mer in textual representation followed by a counter and the
linkage information. The correctness of the reference implementation is checked
by locating each printed k-mer within the original read file. The number of
occurrences is verified as well as the fact that each linkage has to exist at least
once within the reads. The use of k-pairs has to be considered, hence for any
printed k-mer the occurrences of its reverse complement have to be considered
as well.

Running the FPGA design and the reference implementation consecutively, the
outputs of both can be compared for differences. No differences indicate that
the FPGA design produces the same results as the reference design.
4.2 Implementation
CHAPTER 5

Comparison and Evaluation

The first part of this chapter discusses figures that are recorded during evaluation runs of the reference implementation. It is explained why they are recorded and which implications can be made by interpretation of the results. The second part presents evaluations of several individually important aspects, e.g., load-balancing or pre-corrected data. Third, performance estimations are calculated in terms of execution times and conservatively compared to an existing assembler. Last, a concluding discussion on all evaluated aspects is presented.

Four central points are highlighted during this chapter. These points are listed below together with some questions refining the purpose of the specific aspect.

**Workload Distribution:** Are all FPGAs responsible for roughly the same amount of work? Or does one FPGA require far more time while the others are long finished?

**Execution Time:** How long does the graph creation take? This has to be estimated using the reference implementation.

**Memory Utilization:** Is the memory structure, divided into a prefix memory, a suffix memory, and a binary search tree, suitable? How does the behavior/execution time change when memory depth or width is changed?

**Scalability:** How does the implementation behave with increasing numbers of reads? Here, execution time as well as memory utilization need to be considered.

To support the argumentations with figures the following data is recorded during runs of the reference implementation.

**Number of Reads, K-mers:** The overall number of reads and the number of distinct k-mers (in the following called *k-mer nodes*) is recorded. This allows comparison between different data sets in terms of expected effort. Likewise, the impacts of pre-corrected read data can be inspected.
5 Comparison and Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Rhodobacter</th>
<th>Bordetella</th>
<th>Human Chr14</th>
</tr>
</thead>
<tbody>
<tr>
<td>read length</td>
<td>101</td>
<td>36</td>
<td>101</td>
</tr>
<tr>
<td>no. of reads</td>
<td>2,050,868</td>
<td>12,549,138</td>
<td>36,504,800</td>
</tr>
<tr>
<td>coverage</td>
<td>45×</td>
<td>111×</td>
<td>42×</td>
</tr>
<tr>
<td>23-mers</td>
<td>161,705,337</td>
<td>175,687,932</td>
<td>2,883,889,070</td>
</tr>
<tr>
<td>corrected</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no. of reads</td>
<td>1,596,328</td>
<td>10,890,112</td>
<td>32,621,862</td>
</tr>
<tr>
<td>23-mers</td>
<td>83,935,577</td>
<td>151,567,231</td>
<td>2,332,436,273</td>
</tr>
</tbody>
</table>

Table 5.1: Read data sets used for evaluations. The figures in the bottom two lines result from applying the Quake [KSS10] error correction algorithm to the read data set. 23-mer counts are non-distinct.

**Memory Read/Write Requests:** The number of read or write requests to the memory is recorded, i.e., each time a 64 bit word is read or written from or to the memory, respectively. The number of read requests directly correlates with execution time and allows an estimation of the FPGA-based implementation’s execution time. The number of write requests equals the aggregated sum of the k-mer’s frequency counters and therefore is a measure for the workload that was assigned to a certain FPGA.

**Tree Memory Height/Mean Row Fills:** Allows to estimate the balance between the three types of memory. Full prefix rows and empty suffix rows indicate an imbalance in workload – the prefix rows can be shortened.

**Un-/Occupied Memory Rows:** The number of occupied rows within each FPGA’s memory is recorded. It gives insights into how much memory remains unused, or in negative terms is wasted.

**Read Data**

Three sets of genomic read data are used for evaluations. Two bacteria *Rhodobacter sphaeroides* and *Bordetella*, which produce an amount of non-distinct 23-mers in the range of 161 and 175 million. They differ in read length (101 vs. 36) and coverage (45-fold vs 111-fold). Additionally, the chromosome 14 of the human genome is used, which produces about 2.9 billion non-distinct 23-mers.

The evaluations conducted here do not aim to assess quality in any means. They only serve as first insights into possible execution times of the performed graph creation. The bacteria serve as a comparison of two sets of the same size with different properties (read length and coverage). The human chromosome 14 represents an initial indicator for larger genomes. Its size is roughly one thirtieth of the human genome.
\[\text{Table 5.2: Overview of results produced by evaluation runs using the reference implementation.}\]

The data of Rhodobacter and the human chromosome 14 are downloaded from the website\(^1\) of the GAGE study. Bordetella has the accession number ERR007648 in the Sequence Read Archive (SRA)\(^2\) of the NCBI.

\textbf{Overview}

Table 5.3 shows a first overview of some figures resulting from evaluation runs. A de Bruijn graph representation of the above mentioned data sets is created using the reference implementation discussed in Section 4.2.

There are mainly three points that can be derived from the figures. First, corrected read data (indicated by a trailing \texttt{.cor}) is very likely to produce better results. For instance the number of corrected k-mers of the Rhodobacter genome is less than the number of uncorrected k-mers. However, the number of performed read requests reduces even faster, see the Sent/RRQs ratio. Also, the standard deviation of the performed read requests over all FPGAs is reduced significantly.

Second, an even distribution of the workload improves the overall performance. Consider the Rhodobacter\texttt{.equi} figures, which result from an evaluation run with equidistant prefix ranges. The number of read requests is two times higher than the results from a run with optimized prefix ranges.

Third, already the nature of genomic read data can influence the results. Comparing the results of the Bordetella genome (corrected and uncorrected) to the Rhodobacter genome, it can be seen that for Bordetella the number of read requests and the standard deviation reduces far less than for Rhodobacter, even though the number of sent k-mers is roughly the same.

The evaluation runs are conducted for a \(k\) of 23 bases. The k-mers are split up into a prefix of 28 bit, and a suffix of 18 bit. The memory depths are selected accordingly (twice the required size for non-equidistant ranges). For the row widths of RIVYERA’s memory dimensions serve as basis, hence a prefix memory row can hold 32 nodes, a suffix memory row can hold 64 nodes. The following sections discuss and verify the previously stated observations in further detail.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|c|}
\hline
Genome & K-mers Sent & RRQs & Std. Dev. & Sent/RRQs \\
\hline
Rhodobacter\texttt{.cor} & 83,935,577 & 2,308,312 & 314,365 & 35.4 \\
Rhodobacter & 161,705,337 & 35,088,127 & 7,129,070 & 2.8 \\
Rhodobacter\texttt{.equi} & 161,705,337 & 76,794,760 & 17,836,900 & 1.2 \\
Bordetella\texttt{.cor} & 151,567,231 & 3,259,793 & 462,652 & 46.5 \\
Bordetella & 175,687,932 & 5,197,379 & 836,236 & 33.8 \\
\hline
\end{tabular}
\end{table}

\(^1\)http://gage.cbcb.umd.edu/data/index.html
\(^2\)http://www.ncbi.nlm.nih.gov/sra
5 Comparison and Evaluation

Load-Balancing

As mentioned in Section 4.1, a load balancing algorithm is used to calculate an optimal k-mer distribution across the available FPGAs based on the k-mers’ prefixes. Figure 5.1 compares results from graph creation runs with and without load balancing to each other. Figure 5.1a shows the difference in executed write requests to the memory, which directly corresponds to the number of k-mers that are assigned to a specific FPGA. The more balanced the write requests are, the better the load balancing is. Without load balancing, one FPGA performs 6.5 million writes, opposed to a different FPGA performing 75 thousand writes (which is the minimum). With load balancing all FPGAs perform roughly the same amount of writes, but never more than 1.37 million. Obviously, there are certain FPGAs which execute fewer write requests than others. This results from the fact that one FPGA can only take a limited interval of prefixes. However, as discussed in Section 4.1, the created distribution is optimal with regards to the stated requirements.

Figure 5.1b presents the number of created k-mer nodes. If a k-mer occurs twice, it will map to the same node in the memory and increases the node’s counter. It can be seen that the highest k-mer count on one FPGA is reduced by a factor of 3.4 from 3.3 million to 0.97 million. An even amount of nodes on each FPGA increases the probability that the number of queries performed by graph processing algorithms are also more balanced over all FPGAs.

Figure 5.1c shows the immediate impact of load balancing on the execution time, i.e., on the number of performed read requests. To be precise, the diagram contains the maximum value of either the read request to the prefix memory, or the read requests to the suffix and tree memory. It can be seen, that even with load balancing there is a noticeable peak at one FPGA. The peak (35.5 million) is 2.2 times smaller than the peak without load balancing (76.8 million). Considering the fact that the amount of assigned k-mers to the peaking FPGA is exactly the optimal amount shows that this cannot be further optimized by load-balancing. It results from biological diversity in the suffix of a certain prefix yielding full rows for that specific prefix.
(a) Performed write requests to the memory.

(b) Created k-mer nodes.

(c) Performed read requests to the memory.

Figure 5.1: Evaluations of the Rhodobacter bacteria (non-corrected data) showing the impact of the application of a load balancing algorithm (blue) and no load balancing (green). Each bar of the x-axis represents one of 128 FPGAs.
5 Comparison and Evaluation

Pre-Corrected Data

In Section 2.3 a pre-processing step is mentioned that performs error-correcting algorithms prior to graph creation. A lot of single-base errors resulting from the sequencing process can be detected and either corrected or removed using simpler and faster methods than building a de Bruijn graph and processing the graph. Liu et al. [LSM11] found out that this step is important especially for large genomes. It allows to cope with the large memory-requirements of the de Bruijn graph structure.

Figure 5.2 compares results of the graph creation using original genome data to the results from pre-corrected data. For the pre-correction the Quake [KSS10] algorithm is used.

Figure 5.2a shows the difference in the overall created nodes. It can clearly be seen that error corrected data significantly reduces the number of created nodes (on average by a factor of 8). This effect correlates with the inherent purpose of error correction. Many k-mer nodes are created due to sequencing errors in the reads. These k-mers likely occur with low multiplicity but still take a spot in the memory rows resulting in longer chains of read requests when placing a new node or querying for a node.

Figure 5.2b contrasts the amount of k-mers that are placed in either the suffix memory or the tree memory. The results show that for the corrected data the suffix and tree memory contain way fewer nodes than before. Some of the FPGAs do not require them at all. Again, this is expected behavior as long rows are produced by many permutations with the same prefix, i.e., reducing base errors reduces the number of permutations. The empty suffix memory for corrected data is explained by the smallness of the Rhodobacter genome. Corrected data of the human chromosome 14 requires both the suffix memory and the tree memory.

In Figure 5.2c the number of overall performed read requests can be seen. The peak read requests are reduced by a factor of 14.8. Furthermore, the amount of read requests on the different FPGAs is much more balanced.
Figure 5.2: The diagram shows the differences of error-corrected (blue) data to original data (green) of the Rhodobacter bacteria. Each x-axis bar represents one of 128 FPGAs.
5 Comparison and Evaluation

Memory Consumption

Figure 5.3 shows an analysis of the used memory rows per FPGA. Rows are addressed using the prefix (or suffix) of a k-mer. Due to the diverse nature of genomic read data, it is expected that not all rows are used. In Figure 5.3a the percentage of the used rows per FPGA can be seen for the Human Chromosome 14. For the prefix memory an average of 36% of the rows are used, 30% for the suffix memory. The used rows of the prefix memory contain only 4.5 k-mer nodes each (on average), 2.3 k-mer nodes for the suffix memory. Consequently, only 5% of the available prefix memory is used, even less suffix memory.

The impact of using corrected read data can be seen in Figure 5.3c. While the suffix memory rows are rarely used at all (below 5% with roughly 1.6 k-mer nodes per row), also the usage of the overall prefix memory drops below 1% (16% rows with an average of 2 nodes). Nevertheless, this sparseness is traded against access speed to k-mer nodes. Considering, most rows only contain an average of 2 nodes, these nodes can be accessed within two requests to the memory.

Section 5 (Memory Arrangement) presents further inspections on how to restructure the memory, not only to gain better execution times, but also to reduce the waste of memory while preserving fast access speed.
(a) Percentage of used rows on each FPGA. Blue bars represent prefix memory rows while green bars represent rows of the suffix memory.

(b) Average number of k-mer nodes within an occupied row (ignoring unused rows). Blue bars represent prefix memory rows, which are limited to a maximum of 32 nodes. Green bars represent suffix memory rows, which are bound to 64 nodes.

(c) Same data as (a) with the difference that Quake corrected reads are used.

Figure 5.3: The charts show the memory usage on each FPGA after the graph creation for the Human Chromosome 14. The x-axis holds the 128 FPGAs.
5 Comparison and Evaluation

Memory Arrangement

Figure 5.4 shows the results of evaluation runs with different row sizes of the prefix and suffix memory, i.e., a row of the prefix memory can hold up to 32 nodes during one run and only 8 nodes during another run. The sizes of the prefix and suffix themselves are kept constant. The red line results from a prefix memory width of 32 (meaning each row can hold 32 nodes) and a suffix memory width of 64, which is the first naive setup the previous evaluations are performed with. The blue line represents a memory with a width of 8 for both the prefix memory and the suffix memory. Likewise, the green line represents a width of 4 for both memories. In the following, the three different setups are referred to as red, blue, and green.

Figure 5.4a contrasts the amount of read requests to the different memory setups. On the first sight, the green setup seems to have the lowest number of read requests for all FPGA s. But for the 0th FPGA the green setup exceeds the blue setup in the number of read requests. Thus, a smaller row width always yields less read requests for prefix memory and suffix memory. However, if too many nodes end up in the tree memory, the overall number of read request might exceed the benefit from short memory rows.

This fact can also be observed in Figure 5.4b, which holds the number of k-mer nodes that are created in the tree memory. Naturally, the smaller the memory rows, the more k-mer nodes are created in the tree memory. For the 0th and the green setup these are roughly 220,000 nodes. Considering, that a node in the tree memory requires 128 bit, the total memory consumption of these nodes is about 3.4 MB. Space-wise this is feasible (the targeted RIVYERA would provide 128 MBs for the tree memory). However, the increased amount of read requests to the tree memory favors the blue setup in terms of overall performance.

Figure 5.4c shows the number of occupied rows in the suffix memory. As expected, the number of occupied suffix memory rows increases with a decreasing row width of the prefix memory. The thin black line indicates the number of occupied rows in the prefix memory. These figures are the same for either setup as neither the prefix width nor the data changes.

Concluding, it can be said that the first naive choice of memory row widths can be improved. A row width of 8 for both memory types seems to be far more promising than the original widths of 32 and 64. A reduction of the row width implies less space for each prefix. However, with constant memory the prefix can be enlarged further, probably by reducing execution time. Here, the prefix was kept constant for a good comparability. It is also thinkable to increase the suffix used to address the rows of the suffix memory, i.e., having an overlapping prefix and suffix. As Figure 5.4c suggests, the memory hierarchy compensates smaller memory rows by better distributing the nodes among the memory types.
Figure 5.4: The charts show the impacts of changing prefix and suffix memory’s row depths. The red curve indicates the standard setup (pw=32, sw=64). The blue represents a width of 8 for both, likewise indicates green a width of 4 for both memories. X-axis represents 128 FPGAs. The curves do not indicate any correlations between the measuring points but serve for better visibility.
Scalability (Reads)

An important question is how the presented approach behaves with increasing numbers of reads. Figure 5.5 shows results from an evaluation where the measurements are conducted each time one million reads have been processed. This is done for the read data of the Human Chromosome 14, once for corrected data, and once for uncorrected data.

As seen in Figure 5.5a, uncorrected data results in a continuously increasing number of k-mer nodes while the number of k-mer nodes created for corrected read data converges to a near-constant value. After six million reads the number of nodes remains at roughly 700,000.

The performed read requests behave in a similar fashion. Although both curves in Figure 5.5b seem to grow linearly, the read requests of the uncorrected data (blue curve) grow by a factor of 14 between two measuring points while the corrected data’s curve increases by a factor of 7. Furthermore, the blue curve shows a slight increase in slope.

The last observation is supported by Figure 5.5c. Each curve holds the ratio of new read requests to new write requests that were performed in between two measuring points. While this ratio stays constant for corrected data (green curve), it increases for uncorrected data (blue curve).

Concluding, it can be said that the impact of pre-corrected data on created k-mer nodes and read requests increases with the amount of read data. For small genomes (below 3 million reads) there is not much of a difference. However, the more k-mer nodes are created, the likelier it becomes that those need to be placed in the tree memory, yielding more read requests. An error correction of read data requires some time itself, hence it should be evaluated at which point the reduced execution time of corrected reads compensates the effort for error correction.
(a) Number of k-mer nodes created. The blue curve represents uncorrected reads, the green curve corrected reads.

(b) The number of performed read requests. Curves as in (a).

(c) Each measuring point shows the ratio of newly performed read requests to newly performed write requests (newly in regards to the last one million reads). Curves as in (a).

Figure 5.5: The diagrams present the behavior of the graph creation with relation to increasing numbers of reads. After each one million reads of the Human Chromosome 14 results are taken. The x-axis represents the number of reads already processed.
Comparison and Evaluation

Figure 5.6: The diagram presents the number of read requests performed by an FPGA in the presence of different setups of RIVYERA with load-balancing. Blue, red, and green bars represent a setup with 64, 128, and 256 FPGAs, respectively. The x-axis represents the FPGAs.

Scalability (FPGAs)

All previous evaluations are conducted using a setup with 128 FPGAs. Since RIVYERA is available with various numbers of FPGAs, it is examined how the number of FPGAs affects the execution time. Figure 5.6 shows the number of read requests per FPGA (again, the maximum of prefix memory and the sum of suffix memory and tree memory is used). The blue, red, and green bars represent a RIVYERA with 64, 128, 256 FPGAs, respectively. For 128 FPGAs, the figures equal Figure 5.1c. To allow comparisons between the three different setups, the prefix width is kept constant. This implicates that the FPGAs of the smallest setup have to provide enough memory to hold the required amount of prefixes, i.e., in this case, 64 FPGAs require twice the amount of memory as 128 FPGAs. On the other hand, for a setup with 256 FPGAs, the prefix can either be increased, or the amount of memory can be halved. However, with relation to scalability the available memory per FPGA is irrelevant, as long as all required prefixes can be stored.

As seen in Figure 5.6 the blue, red, and green bars have their peak at FPGA 43, 61, and 114, respectively. In terms of read requests the peaks are at 82 million, 35 million, and 18 million read requests, respectively. This indicates that doubling the number of FPGAs halves the required execution time of the graph creation. Considering no dependencies exist between any two FPGAs, this finding was expected. However, the number of read requests performed by FPGA 0 decreases hardly when increasing the number of FPGAs. Thus, this FPGA might prevent a further decrease in execution time for more than 256 FPGAs. However, this behavior likely results from the fact that the implementation replaces any $N$ within the read data by an $A$. This yields a large numbers of prefixes primarily
5.1 Performance Evaluation

<table>
<thead>
<tr>
<th>Genome</th>
<th>LBRs</th>
<th>RRQs</th>
<th>WRQs</th>
<th>Cycles</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rhodobacter_cor</td>
<td>8</td>
<td>2,308,312</td>
<td>697,236</td>
<td>33,277,632</td>
<td>0.3</td>
</tr>
<tr>
<td>Rhodobacter</td>
<td>2</td>
<td>35,088,127</td>
<td>1,360,474</td>
<td>431,941,316</td>
<td>4.3</td>
</tr>
<tr>
<td>Bordetella_cor</td>
<td>2</td>
<td>3,259,793</td>
<td>1,235,852</td>
<td>49,004,332</td>
<td>0.5</td>
</tr>
<tr>
<td>Bordetella</td>
<td>7</td>
<td>5,197,379</td>
<td>1,696,795</td>
<td>74,883,104</td>
<td>0.75</td>
</tr>
<tr>
<td>HgChr14_cor</td>
<td>2</td>
<td>203,270,973</td>
<td>18,611,883</td>
<td>2,588,146,740</td>
<td>25.9</td>
</tr>
<tr>
<td>HgChr14</td>
<td>3</td>
<td>419,552,519</td>
<td>22,879,438</td>
<td>5,217,665,732</td>
<td>52.2</td>
</tr>
<tr>
<td>HgChr14_cor 8×8</td>
<td></td>
<td>93,840,769</td>
<td>18,618,691</td>
<td>1,275,038,756</td>
<td>12.8</td>
</tr>
<tr>
<td>HgChr14 8×8</td>
<td></td>
<td>157,798,941</td>
<td>22,951,152</td>
<td>2,077,196,508</td>
<td>20.8</td>
</tr>
</tbody>
</table>

Table 5.3: Estimated execution times of the graph creation for the different read data sets. A trailing _cor indicates error-corrected data using the Quake algorithm. Results marked with a 8×8 are performed with different memory row widths. LBRs (Load-balancing rounds) names the number of loops of the load balancing algorithm.

As mentioned before, no runs on the actual RIVYERA platform could have been performed. Thus, the performance in terms of execution time can only be estimated. For this purpose, the slowest element of the graph creation pipeline is identified. It showed that the access to the RAM states this component. The time required to access a random cell in the RAM is measured as well as the time to write a word to the memory.

The number of read and write requests has been determined by the evaluations presented in the sections above. To get an estimation of the overall execution times one can consider the clock frequency, the FPGA with the highest memory accesses, and the number of cycles to access the memory.

Execution Times

To estimate the execution time of an FPGA the required clock cycles of the memory interface are counted. For a read request 12 cycles are required from the time the request is placed in the interface’s FIFO until the requested data is available. Requesting more than one word at once yields two additional cycles per word. Writing a word to the memory takes 8 cycles with one additional cycle per consecutive word. Using these numbers a lower bound for the execution time can be determined. The execution time of the overall graph creation is simply the maximum of all the FPGAs’ execution times, since they all run in parallel.
Every read request (RRQ) that is performed by the memory accessing component demands exactly one word, thus, each request lasts 12 cycles. The same goes for the write requests (WRQ), yielding 8 cycles per request. Assuming that the input FIFO of the memory accessing component offers k-mers each time a new one is required (the memory component is busy each cycle), the overall cycles can be calculated by $RRQs \times 12 + WRQs \times 8$. The current implementation runs with 100 MHz, accordingly, the time is calculated by multiplying with $10^{-8}$.

Table 5.3 presents the execution time estimations for the evaluated read data sets. For each genome the maximum of read and write requests is given, the corresponding number of cycles is calculated, and also expressed in terms of seconds. It can be seen that the estimated time for the graph creation of the two smaller genomes is below one second for pre-corrected data. The human chromosome 14 takes less than a minute with and without pre-corrected data. Adapting the memory row sizes of prefix memory and suffix memory to 8 nodes per row results in halved execution times (12.8 seconds opposed to 25.9 seconds for corrected data).

Furthermore, it showed that the number of rounds of the load-balancing algorithm and the discussed adaption strategy, presented in Section 4.1, is equal to or below eight rounds for each of the used read data sets.

Finally, it should be remarked, that the presented figures are estimated lower bounds. Neither overhead is considered nor the fact that due to the nature of the genome data some FPGAs will have idle times while only data for different FPGAs is broadcasted. Nevertheless, real execution times should settle not too far from the estimated figures.

### Comparison

A comparison to other assemblers lacks in several ways. First, only the creation of a de Bruijn graph is implemented, missing the crucial steps of contig creation and scaffolding to perform an overall de novo genome assembly. Thus, only the time required to build the graph can be compared to other assemblers. The result of this gives a hint on the overall performance but has to be treated with caution. A data structure that can be constructed in a fast and efficient way might be slow when it comes to accessing the data again, especially, considering that it is a distributed structure. Furthermore, not all de novo assemblers serve as a valid object of comparison. SOAPdenovo uses a shared memory and opposed to PASHA requires more time to create the graph than the amount of time spent for graph processing. Likewise, it is hard to figure out the actual execution time that an assembler requires to create the graph.

#### PASHA

The publication by Liu et al. [LSM11] presents several figures that allow a rather speculative estimation of the time spent for graph creation. They stated how long each of the assembly steps (graph creation, contig generation, scaffolding, etc.) for the human genome takes. The graph construction and
simplification is listed as one step. Hence, the estimation of 10% of the time spent for plain graph creation is vague. However, considering that the overall assembly of the Bordetella genome requires roughly 110 seconds, about 10 seconds are required for graph construction (performed on 32 CPU cores, the authors do not observed a significant decrease in execution time by increasing the number of cores any further). Comparing this time to the estimated run times in Table 5.3, the approach presented in this thesis would be more than 10 times faster.

PASHA is the fastest assembler in terms of execution time of the assemblers presented in Section 2.4 despite the SparseAssembler. SparseAssembler pursues a different approach by storing only a fraction of the generated k-mers – a method that might be incorporated into the implementation of this thesis. Overall, the estimated execution times are promising and satisfying for the current state of progress.

**Quality**

No statements can be made with regards to the quality of the assembly results. This thesis developed an approach to construct the data structure that will be used by algorithms that directly impact the quality. The available information extracted from the genomic read data and hold in the data structure equals the information used by other well-performing assemblers (e.g., PASHA and ABySS). Hence, the data should be sufficient to perform further steps of the assembly pipeline.
5 Comparison and Evaluation
CHAPTER 6

Conclusion

This thesis presents an approach to perform the creation of a distributed de Bruijn graph used for the de novo genome assembly on the hardware platform RIVYERA. First, basic knowledge in the field of bioinformatics and state-of-the-art technologies is introduced in the area of genome sequencing and genome assembly. Several closely related de novo genome assemblers are presented along with existing comparisons and evaluations of these assemblers. The properties of RIVYERA and its architecture are discussed, along with a short introduction into usage paradigms.

The main contribution of this thesis is a highly optimized data structure used for a de Bruijn graph creation on RIVYERA. Despite the high level of optimization and customization, its general ideas can be reused elsewhere. Among these ideas are implicit addressing using a prefix of a k-mer, construction of a memory hierarchy to allow efficient node lookup, usage of k-pairs to save half the memory requirements, and a load-balancing across the FPGAs based on the counts of the k-mers’ prefixes.

The previous chapter presents the results of detailed evaluations of the approach’s central aspects. Here, the importance and the impacts of the aforementioned strategies become clear. For instance, the memory hierarchy along with a load-balancing across the FPGAs improves performance significantly.

Overall, the evaluations show promising results with regards to the execution times of the graph creation. Using 128 FPGAs, speedups in the order of 10 are expected in comparison with PASHA, one of the fastest state-of-the-art assemblers. The approach scales well with increasing numbers of FPGAs, e.g., by doubling the number of FPGAs to 256 the required execution time is halved.

Furthermore, most of the latest assemblers require supercomputers or large computation clusters. These are expensive in both acquisition and operation costs. Here, RIVYERA is superior particularly with regards to operation costs. The quality of an assembled genome cannot be assessed in this work since only a part of the overall assembly pipeline is performed. However, the quality is expected to roughly match the quality that other assemblers produce, as the quality rather depends on the error correction algorithms than on the underlying data structure.
6 Conclusion

The important procedure of load-balancing can be completely incorporated into previous pipeline steps (pre-error-correction) without extra effort in time. Besides, further improvements of the currently used setup are possible, e.g., an adaption of the row widths of the prefix and suffix memory decreased the execution times by a factor of 2. A fully equipped RIVYERA will provide deeper insights into these facets for larger genomes. The evaluations presented indicate that human-sized genomes can be handled easily.

RIVYERA’s architecture is ideal to create a distributed de Bruijn graph across all available FPGAs. The task can be split into independent sub-tasks performed by each FPGA, requiring little logic to be implemented. Thus, there are sufficient resources left to add additional implementations for further steps of the assembly pipeline. Examples are algorithms for tip removal and bubble merging. However, these steps require communication between the different FPGAs. The fact that the communication can be realized efficiently needs to be proved. Lastly, the essential facts of this thesis are listed in a recapitulating form.

- Distributed memory and distributed de Bruijn graph.
- K-pairs combine a k-mer and its reverse complement.
- Rearranging the memory to free half of it (Remapping k-pairs).
- K-mers split into prefix and suffix.
- Memory hierarchy split into prefix memory, suffix memory, and tree memory.
- Implicit addressing using either the prefix or the suffix.
- Load-balancing across available FPGAs (incorporated into pre-correction).

6.1 Future Work

In the following, several aspects that are dedicated in upcoming work are discussed separately.

FPGA Design As mentioned earlier, an FPGA design of the discussed graph creation approach exists but is untested with relation to the memory accessing components.

Once a fully working and tested FPGA design is completed, further improvements can be made. For instance, the prefix memory has to cope with the most requests since every request starts here (even if it ends up in the suffix memory or the tree memory). Hence, using two components to handle requests to the prefix memory would yield a further speedup. This fits RIVYERA’s architecture as an additional memory port is available for the prefix memory.
Furthermore, it will be easier to determine values for the prefix size and for the memory row widths of prefix memory and suffix memory, such that an optimal execution time is reached.

**Overall De Novo Assembly Pipeline**  Section 2.3 describes the overall process of the de novo assembly. This includes more stages than the graph creation presented in this thesis. Further work includes finding solutions to perform the missing parts of the assembly pipeline using RIVYERA and to incorporate all steps into one pipeline.

**Evaluation of Large Genomes**  The read data set of the human chromosome 14 used for the evaluations in Chapter 5 represents only a subset of the overall human genome. Salzberg et al. [SPZ+12] created the data set by extracting all reads from a data set of a human genome that belong to the 14th chromosome. The authors state that this is about one thirtieth of the overall human genome. Considering this, the available resources of RIVYERA are able to hold the whole human genome. However, further evaluations of this size have to be conducted with a fully equipped RIVYERA since the reference implementation’s performance is not feasible to provide exhaustive evaluations.

**Third-Generation Sequencing**  As stated in Section 4.2 currently only reads up to a length of 128 bases are supported. Third-generation sequencing technologies will create longer reads, allowing assemblies of higher quality. The FPGA design allows to incorporate longer reads as soon as they are required. Different sizes of the \( k \) should be considered as well. Li et al. [LZR+10] found 23 to be the best choice for short reads of second-generation sequencing technologies. This value will probably increase for third-generation sequencing.
6 Conclusion
1 Introduction

1.1 Purpose of this Thesis ........................................ 3
Contributions .................................................. 4
1.2 Outline ..................................................... 4

2 Basics of Bioinformatics ........................................ 7

2.1 Biological Background ...................................... 7
2.2 Genome Sequencing ......................................... 11
2.3 Genome Assembly ........................................... 12
2.4 Assembly Strategies ........................................ 17
  EULER – Pevzner et al. [PT01] .............................. 17
  Velvet – Zerbino et al. [ZB08] ................................. 18
  ABySS – Simpson et al. [SWJ09] ............................ 19
  ALLPATHS – MacCallum et al. [MPG09] ..................... 20
  SOAPdenovo – Li et al. [LZR10] ............................. 21
  Ray – Boisvert et al. [BLC10] ................................. 21
  PASHA – Lui et al. [LSM11] ................................. 21
  Conway and Bromage [CB11] ................................. 22
  SparseAssembler – Ye et al. [YMC12] ......................... 22
  SGA – Simpson and Durbin [SD12] ......................... 23

2.5 Assembler Comparison ....................................... 25
  Assemblathon 1 – Earl et al. [EBS11] ....................... 25
  Narzisi and Mishra [NM11] .................................. 25
  Alkan et al. [ASE11] ......................................... 26
  Salzberg et al. [SPZ12] ..................................... 26

3 RIVYERA .................................................. 29

3.1 FPGAs ..................................................... 29
Development .................................................... 30
Benefits ........................................................ 32
3.2 Architecture ................................................ 32
Host ............................................................ 33
3.3 Communication API ........................................ 33
4 De Novo Genome Assembly on RIVYERA 37
  4.1 Concept and Data Structure 37
     Concept 38
     Node Representation 40
     Optimizing Memory Structure 40
     Addressing 42
     Memory Hierarchy 43
     Load Balancing 44
     K-mer Queries and Graph Traversal 48
     Alternative Data Structures 49
  4.2 Implementation 50
     Overview 51
     Control Application 51
     FPGA Design 53
     Reference Design 56
     Validation 57

5 Comparison and Evaluation 61
  Read Data 62
  Overview 63
  Load-Balancing 64
  Pre-Corrected Data 66
  Memory Consumption 68
  Memory Arrangement 70
  Scalability (Reads) 72
  Scalability (FPGAs) 74
  5.1 Performance Evaluation 75
     Execution Times 75
     Comparison 76
     Quality 77

6 Conclusion 79
  6.1 Future Work 80

Detailed Tables of Content 84

Bibliography 93
**Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BRAM</td>
<td>Block RAM</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable Logic Block</td>
</tr>
<tr>
<td>CMT</td>
<td>Clock Management Tile</td>
</tr>
<tr>
<td>COPACOBANA</td>
<td>Cost-Optimized Parallel Code Breaker</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DDR-RAM</td>
<td>Double Data Rate RAM</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>DNA</td>
<td>Desoxyribonucleic Acid</td>
</tr>
<tr>
<td>FRC</td>
<td>Feature-Response Curve</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In-First-Out</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GAGE</td>
<td>Genome Assembly Gold-Standard Evaluations</td>
</tr>
<tr>
<td>GOLD</td>
<td>Genomes Online Database</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>NCBI</td>
<td>National Center for Biotechnology Information</td>
</tr>
<tr>
<td>OLC</td>
<td>Overlap-Layout-Consensus</td>
</tr>
<tr>
<td>PCIe</td>
<td>Peripheral Component Interconnect Express</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
</tbody>
</table>
**Detailed Content**

**RAM**  Random Access Memory  
**RNA**  Ribonucleic Acid  
**mRNA**  messenger RNA  
**RIVYERA**  Reconfigure Versatilely Your Efficient Raw Architecture  
**SBH**  Sequencing by Hybridization  
**SGA**  String Graph Assembler  
**SNP**  Single Nucleotide Polymorphism  
**SRA**  Sequence Read Archive  
**VHDL**  Very High Speed Integrated Circuit Hardware Description Language
List of Figures

1 Introduction

2 Basics of Bioinformatics
  2.1 Illustration of the DNA double helix and its components, the bases. 8
  2.2 Overview of the gene expression process. . . . . . . . . . . . . . . 10
  2.3 Visualization of the genetic code. . . . . . . . . . . . . . . . . . . 11
  2.4 Different sequencing technologies. . . . . . . . . . . . . . . . . . . 13
  2.5 Overview of an assembly pipeline. . . . . . . . . . . . . . . . . . . 15
  2.6 Different de Bruijn graph representations. . . . . . . . . . . . . . 18
  2.7 Frequent error-prone de Bruijn graph structures. . . . . . . . . . 19
  2.8 An overview of several de Bruijn graph representations in different
de novo genome assemblers. . . . . . . . . . . . . . . . . . . . . . . . . . 24

3 RIVYERA
  3.1 Image of RIVYERA’s front panel. . . . . . . . . . . . . . . . . . . . 30
  3.2 Schematic overview of RIVYERA’s architecture. . . . . . . . . . . 32

4 De Novo Genome Assembly on RIVYERA
  4.1 Partition of available memory to FPGAs. . . . . . . . . . . . . . . 39
  4.2 Illustration of a k-mer’s storage to memory. . . . . . . . . . . . . . 40
  4.3 Schematic overview of the memory arrangement. . . . . . . . . . . 41
  4.4 Overview of the memory hierarchy. . . . . . . . . . . . . . . . . . . 43
  4.5 The FPGA design’s architecture. . . . . . . . . . . . . . . . . . . . 51
  4.6 Illustration of the node inserting state machine. . . . . . . . . . . 56

5 Comparison and Evaluation
  5.1 Load-balancing evaluations. . . . . . . . . . . . . . . . . . . . . . . 65
  5.2 Evaluations of pre-corrected read data. . . . . . . . . . . . . . . . 67
  5.3 Memory consumption of different memory types. . . . . . . . . . . 69
  5.4 Impacts of changing memory rows’ sizes. . . . . . . . . . . . . . . 71
  5.5 Assessment of the approach’s scalability (reads). . . . . . . . . . 73
  5.6 Assessment of the approach’s scalability (FPGAs). . . . . . . . . . 74

6 Conclusion

Bibliography
# Listings

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Basics of Bioinformatics</td>
</tr>
<tr>
<td>3</td>
<td>RIVYERA</td>
</tr>
<tr>
<td>3.1</td>
<td>Short VHDL code example.</td>
</tr>
<tr>
<td>3.2</td>
<td>A typical workflow of a host application controlling RIVYERA.</td>
</tr>
<tr>
<td>4</td>
<td>De Novo Genome Assembly on RIVYERA</td>
</tr>
<tr>
<td>4.1</td>
<td>Load-balancing algorithm used to balance the number of handled k-mers per FPGA as good as possible.</td>
</tr>
<tr>
<td>4.2</td>
<td>Pseudo code of the host program.</td>
</tr>
<tr>
<td>5</td>
<td>Comparison and Evaluation</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
</tr>
<tr>
<td>Bibliography</td>
<td>93</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Basics of Bioinformatics</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2.1 Comparison of the discussed de novo assemblers with focus on their interpretation and representation of a de Bruijn graph.</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>RIVYERA</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>De Novo Genome Assembly on RIVYERA</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>4.1 Resource utilization of the current FPGA design.</td>
<td>57</td>
</tr>
<tr>
<td>5</td>
<td>Comparison and Evaluation</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>5.1 Read data sets used for evaluations.</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>5.2 Overview of results produced by evaluation runs using the reference implementation.</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>5.3 Estimated execution times of the graph creation for the different read data sets.</td>
<td>75</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>79</td>
</tr>
<tr>
<td>Bibliography</td>
<td>93</td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


[HWA+03] Xiaoqiu Huang, Jianmin Wang, Srinivas Aluru, Shiaw-Pyng Yang, and LaDeana Hillier. PCAP: a whole-genome assembly program. *Genome research*, 13(9):2164–70, September 2003. 16

Bibliography


[KPP+06] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, Andy Rupp, Manfred Schimmler, and G Horst. How to Break DES for EURO 8,98. SHARCS, 20, 2006. 3, 29


95
Bibliography


[MPG+09] Iain Maccallum, Dariusz Przybylski, Sante Gnerre, Joshua Burton, Ilya Shlyakhter, Andreas Gnirke, Joel Malek, Kevin McKernan, Swati Ranade, Terrance P Shea, Louise Williams, Sarah Young, Chad Nusbaum, and David B Jaffe. ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads. Genome biology, 10(10):R103, January 2009. 20, 25, 83


[WBB+10] Lars Wienbrandt, Stefan Baumgart, Jost Bissel, Carol May Yen Yeo, and Manfred Schimmler. Using the reconfigurable massively parallel


