Master's thesis
Master's Programme in Data Science

GPU accelerating distributed succinct de Bruijn graph construction

Topi Laanti

March 21, 2022

Supervisor(s): Professor Keijo Heljanko

Examiner(s): Professor Keijo Heljanko
Dr. Jarno Alanko

University of Helsinki
Faculty of Science

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Degree programme
Faculty of Science	Master's Programme in Data Science

Tekijä — Författare — Author

Topi Laanti

Työn nimi — Arbetets titel — Title

GPU accelerating distributed succinct de Bruijn graph construction

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidantal — Number of pages
Master's thesis	March 21, 2022	61

Tiivistelmä — Referat — Abstract

The research and methods in the field of computational biology have grown in the last decades, thanks to the availability of biological data. One of the applications in computational biology is genome sequencing or sequence alignment, a method to arrange sequences of, for example, DNA or RNA, to determine regions of similarity between these sequences. Sequence alignment applications include public health purposes, such as monitoring antimicrobial resistance.

Demand for fast sequence alignment has led to the usage of data structures, such as the de Bruijn graph, to store a large amount of information efficiently. De Bruijn graphs are currently one of the top data structures used in indexing genome sequences, and different methods to represent them have been explored. One of these methods is the BOSS data structure, a special case of Wheeler graph index, which uses succinct data structures to represent a de Bruijn graph.

As genomes can take a large amount of space, the construction of succinct de Bruijn graphs is slow. This has led to experimental research on using large-scale cluster engines such as Apache Spark and Graphic Processing Units (GPUs) in genome data processing.

This thesis explores the use of Apache Spark and Spark RAPIDS, a GPU computing library for Apache Spark, in the construction of a succinct de Bruijn graph index from genome sequences. The experimental results indicate that Spark RAPIDS can provide up to 8× speedups to specific operations, but for some other operations has severe limitations that limit its processing power in terms of succinct de Bruijn graph index construction.

ACM Computing Classification System (CCS):
Applied computing → Life and medical sciences → Genomics → Computational genomics
Computing methodologies → Computer graphics → Graphics systems and interfaces → Graphics processors

Avainsanat — Nyckelord — Keywords

Graphics Processing Unit, Apache Spark, Succinct data strucutures, de Bruijn graphs

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1. Introduction

1.1 Motivation

Computational biology as a field has evolved over the last 20 years [11], due to the explosive increase of biological data in terms of volume, velocity and variety that has come with the technological advancement of computing technology. This rapid development has led to the development of many applications that make use of computational biology in the modern world, for example battling the SARS-CoV-2 [18] pandemic. One of the emerging methods used in computational biology is *Sequence alignment*.

Sequence alignment [26] is a way of arranging sequences of DNA, RNA or protein to determine regions of similarity between the sequences using *reads*. These regions of similarity may stem from a consequence of a functional, structural or evolutionary relation between the different sequences. A read is a sequence that is obtained from a fragmented genome, where each sequenced fragment produces a read.

An approximate form of sequence alignment, *pseudoalignment*, gives only the information whether a read matches to a reference sequence or not [22], and is often sufficient for analysis. Compared to pseudoalignment, regular alignment also returns the location of the match in the genome. Pseudoalignment has the advantage of being computationally more efficient in terms of speed and cost, a property that becomes important when working with a large amount of data.

Genome indexes created for sequence alignment have many applications. Metagenomic sequences are for example used for public health purposes, such as monitoring antimicrobial resistance [27]. Antimicrobial resistance is considered to be one of the top public threats, as its spread will lead to increased mortality for many bacterial illnesses [27]. The research into antimicrobial resistance has been made possible due to the increasing amount of sequences available, and data structures that represent it efficiently.

The technology for producing sequences has made remarkable progress in the last 20 years. Large sequencing initiatives have been completed that aim to study genetic variation for humans and agriculturally and bio-medically important species. For example, human genome projects 1000 Genomes was announced in 2008 and completed

in 2015 [4], and the 100 000 Genomes was announced in 2013 and completed in 2018*. These projects showcase the increase of speed of sequencing projects [10].

As sequence alignment is a computationally heavy task, it is important to ensure that tools for it are reliable and efficient. Further development of these tools and methods is important, in terms of both improving software and hardware used in sequence alignment.

Software can be improved by for example by using compact data structures such as the *succinct de Bruijn graph* that enable the usage of fast algorithms that take advantage of its compact form.

Hardware can be improved by the development of various different types of processors and by scaling out and increasing performance by introducing more processors that parallelize the workload. Improving current processors has led to the usage of graphics processing units (GPUs) as an alternative to the traditional central processing units (CPUs). GPUs have been a major factor in modern neural network computing [21] and due to their usefulness in, for example, cryptocurrency mining [16], demand for them has greatly risen†.

1.2 Background

The de Bruijn graph has become a staple in computational biology since it was introduced in the context of bioinformatics [19, 27, 20]. It is widely used in modern genomics, as it is essential in genome assembly [27]. As de Bruijn graphs are often created from large-scale data and have different variations depending on the application, a number of different ways to implement and represent them have been created.

De Bruijn graphs are constructed from a set of strings called *k-mers*, that are substrings of a biological sequence that are the length of k. A set of k-mers can be thought to represent a genome sequence in its entirety, while a de Bruijn graph constructed from a set of k-mers represents the k-mers and the relation between them in a compressed form. A variant of the original de Bruijn graph, the *colored de Bruijn graph*, includes the source of each vertex in the graph as additional information.

The intended use of colored de Bruijn graphs is to represent massive population-level sequence data that has become abundant due to the rapid advancement of sequencing technology in the last 20 years. From these graphs, variant information of an individual or population can be inferred [27, 20]. They are built in various different approaches and representations.

While the initial application [20] for colored de Bruijn graphs was for assembly

*https://www.genomicsengland.co.uk/about-genomics-england/the-100000-genomes-project/
†https://www.bbc.com/news/technology-55755820

and genotyping, it has been used in other fields of computational biology, such as pan-genomics [41].

However, as the datasets are large, the methods to analyze them are limited. While colored de Bruijn graphs have served as a popular choice in computational biology, they are still hindered by few factors [17]. First, building a de Bruijn graph from raw sequence reads is computationally demanding, and consumes a lot of memory. Second, as the application purposes can vary, suitable software libraries for a specific purpose may not exist, meaning that often data structures to index the de Bruijn graph have to be re-implemented. Furthermore, adding colors to the de Bruijn graph accelerates the already problematic memory consumption, as the colors tend to use more memory than the edges and vertices of the graph [17].

As the amount of data is constantly growing, the need to implement the colored de Bruijn graphs using *succinct data structures* has risen. Succinct data structures aim to use as little space as possible while retaining fast operations. Succinct de Bruijn graphs can be used for fast and light pseudoalignments, and as a consequence, the research for space-efficient representations of de Bruijn graphs has grown. The formal descriptions of a de Bruijn graph, succinct de Bruijn graph, colored de Bruijn graph, and k-mers will be introduced in Chapter 2.

Examples of recent research into different succinct representations of de Bruijn graphs include TwoPaCo [25], BCALM2 [9] and BiFrost [17]. TwoPaCo builds a succinct colored de Bruijn graph from k-mers which are either branching or located at the extremities of unitigs, a special kind of set of reads from k-mers, that they call *junction* k-mers [25]. BCALM2 also uses unitigs and partitions to make parallel processing possible, combining the partitions after they have been computed [9]. Bifrost adopts a similar approach, using unitigs, partitioning, and a space-efficient probabilistic data structure called Bloom filter for *minimizer* hashing, to provide a compact form for a succinct de Bruijn graph [17]. Minimizer of a k-mer x is a l-mer y occurring in x such that $l < k$ and y is lexicographically smallest of all the l-mers in x [17].

Similarly to the increasing availability and volume of sequencing data, the amount of data in general has rapidly grown in recent years [35]. Creating tools for accessing, analyzing and storing massive amounts of data has been a popular topic in both research and industry. This had led to the creation of various frameworks suited for these purposes, such as Apache Spark [39]. Apache Spark is a cluster computing framework that enables parallel processing between multiple nodes, while retaining a simple programming interface that does not require the programmer to implement the parallelity. Specifics of Spark will be explored in more detail in Chapter 2.6. While Spark and similar frameworks have wide applications across different fields, they have not had wide use in the analysis of genomic and proteomic sequences [32].

However, several tools exist that make use of Spark and other similar frameworks like MapReduce in the context of computational biology. Hadoop-BAM software library made use of Hadoop distributed computing framework to manipulate sequencing data, and it could also be used to interface legacy bioinformatics data formats in Apache Spark [29]. Software tool KCH was the first suite of MapReduce algorithms specifically designed for linguistic and informational analysis of large collections of biological sequences, implemented in Apache Hadoop [31]. KCH showed that big data technologies can be superior to highly-optimized shared memory multiprocessor approaches, even when dealing with data that is considered mid-size. It has given experimental evidence that big data technologies show promise for being an effective solution of a broad spectrum of problems in computational biology, ranging from basic primitives to analysis and storage pipelines [31].

ADAM, a Spark-based toolkit meant for exploring genomic data, is an early attempt to use Spark to extract k-mers. It is able to process large genomic sequences by splitting sequences into different nodes. Being an early exploration, it suffers from a simplistic approach and thus has a very poor resource utilization [32, 24].

Another example is FastKmer, a system used for the extraction of k-mer statistics from large collection of genomic and meta-genomic sequences [32]. FastKmer is implemented in Spark, and makes use of compression and custom partitioning to optimize the results. While building succinct de Bruijn graphs was not the primary motivation for FastKmer, k-mer statistics can be used to construct de Bruijn graphs. FastKmer highlighted the importance of tuning Spark parameters, and emphasized that naive usage of Spark and similar tools can lead to poor results, but with proper tuning, the results can be impressive.

While distributed computing frameworks have received a lot of attention recently as a solution to big data and heavy computing, different approaches have also been proposed. Graphical Processing Units (GPUs) have massive parallel computing capabilities that enables applications that were previously thought infeasible because of long execution times [28]. GPUs have evolved from a configurable graphics processors to programmable parallel processors, and are a many-core multithreaded multiprocessors that excel at both graphics and computing applications. Modern GPUs have thousands of parallel processor cores that execute tens of thousands of parallel threads, that can be used to solve large problems that have inherent parallelism. They are also cost-effective, and have become the most prevalent massively parallel processing resource available.

The original driving force for GPUs was to render real-time graphics, a computing task which has tremendous inherent parallelism. GPU computing frameworks, such as the CUDA parallel computing model, can be used to create programs written in

C or C++ code that scales transparently over a wide range of parallelism. GPU technology has vastly improved in the last 20 years, with transistor counts and CUDA parallel computing cores doubling every 18 months since their introduction in 2010. In 2007, NVIDIA introduced the Tesla C870, D870 and S870 GPU card, deskside and rack-mount GPU computing systems that contain one, two and four T8 GPUs. These CUDA-capable GPUs enabled researchers and industry developers to create a diverse range of applications. The modern equivalent, the NVIDIA V100 is a massive improvement, for example with the number of CUDA cores rising from the 4×128 of S870 to the 6912 of A100*†.

The interest in GPUs in the field of computational biology has grown, with GPUs being recently experimented in applications involving genome assembly. Gerbil is one of the more notable k-mer counting tools that use GPUs [13]. Written in CUDA, it proved to be noticeably more efficient than a similar CPU oriented solution when the length of k was sufficiently large. It employs the use of supermers, a contiguous sequence of bases wherein each k-mer shares the same minimizer [13]. However, Gerbil is a single-node shared memory system, meaning that it is not practical for analyzing datasets that are massive in scale.

DEDUKT was the first GPU-accelerated distributed-memory parallel k-mer counter [30]. Implemented using CUDA, experimental results show that the GPU-based computation reduced the execution time by one to two orders of magnitude compared to distributed-memory CPU counterpart when using large-scale genomic data. However, the GPU acceleration results in a communication bottleneck, that is addressed by using supermers similarly to ones used in Gerbil. DEDUDKT outperforms existing CPU based framework by up to 150× on a *H. sapien* 54× genome data set, showcasing the massive potential of distributed GPU computing.

Few GPU-based approaches to build succinct de Bruijn graph have also been created with the intention of providing a speedup for succinct de Bruijn graph construction. ParaHash is a succinct de Bruijn graph assembler that partitions the input data into a compact format, and parallelizes the computation between CPUs and GPUs of a single computer [33]. ParaHash tries to make use of all processors available in a single machine to assemble genomes that are too large to fit into memory. However, Parahash is limited due to how it handles repeat k-mers, as it cannot be used for micro-assembly, a form of assembly that improves the accuracy of the graph. GATK HC is another GPU-based succinct de Bruijn graph assembler [34]. It assumes there are no repeat k-mers in its input dataset and calculates the occurrences of $(k+1)$-mers in parallel on the GPU, and only used CPU to check for repeat k-mers. Using these

*https://www.nvidia.com/en-us/data-center/a100/

†https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

techniques, it achieves up to 2.66× speedup compared to CPU implementation, and it is well suited for micro-assembly.

As methods for both parallel distributed computing frameworks, such as Spark, and GPU computing have been on the rise in large-scale computing, attempts to combine both into a single united framework have been made. DEDUKT is an example of how combining both distributed parallel computing and GPUs can lead to massive speedups, but it has the downside of being suited only for a very specific purpose. Spark RAPIDS library is an addition to the Apache Spark framework that enables Spark to use GPU computing*. Built on top of CUDA, Spark RAPIDS can potentially provide a cost-effective and easily programmable speedup solution to a wide array of applications, including genome sequencing.

This thesis explores combining succinct de Bruijn graph construction with parallel computing and GPU computing, and showcases the results of a Spark RAPIDS enabled tool that builds a succinct de Bruijn graph. This tool is modeled after Themisto [22], a pseudoalignment tool created by Alanko et al., that constructs and uses succinct de Bruijn graphs, and is also part of its suite. For the remainder of the thesis, the experimental pipeline explored in this thesis is referred to as Spark Themisto, with it being additionally referred as GPU-enabled Spark Themisto when considering its version that uses RAPIDS, while the original Themisto created by Alanko et al. is referred as original Themisto. However, when discussing simply Themisto, the discussion applies to all versions unless explicitly mentioned otherwise.

1.3 Structure of the Thesis

The rest of the thesis is organized as follows: Chapter 2 gives an overview of the preliminaries related to the topic, presenting a detailed overview of a de Bruijn graph and its variations. Chapter 3 explains the methods used, along with a introduction to Themisto. The results of the experiments are shown in Chapter 4, while Chapter 5 concludes.

*https://www.nvidia.com/en-us/deep-learning-ai/solutions/data-science/apache-spark-3/ebook-sign-up/

2. Preliminaries and definitions

In order to understand the upcoming definitions such as de Bruijn graphs, crucial preliminaries have to be introduced first in order to understand data structures and tools used by Themisto.

A string is in traditional computer programming a sequence of characters. A string X can be defined as $T[0...n-1] = t[0]t[1]...t[n-1]$, n being the length of the string, where individual $t[i]$ represents a single character, where $0 \leq i \leq n-1$. Each character belongs to an ordered alphabet Σ that is the size of σ. A substring of T is defined as $t[i]t[i+1]...t[j]$, where $0 \leq i \leq j \leq n-1$. Now, prefix of T is defined as a substring where $i = 0$, and suffix of T is defined as a substring where $j = n-1$. An empty string that contains no characters is denoted with ϵ, and is the prefix and suffix of any possible string. If a string is the length of k, it is called a k-mer.

A set of strings can be ordered by lexicographic order and colexicographic order. The lexicographic order of strings is given by reading strings from left to right, where the order of strings is determined by the first character that differs and their respective ordering in the alphabet. Colexicographic ordering is the same except instead of reading the string from left to right, the string is read from right to left. If strings being ordered have lengths that differ, a padding of null characters can be added to either to the beginning or to the end, depending on the ordering used. These null characters are treated as smaller than any other element in alphabet Σ. This enables the comparing of strings with different lengths. For example, if $\Sigma = \{a, b, c\}$, where the alphabet Σ is ordered in the following way: $a < b < c$, the lexicographic order of a set of strings $\{ac, a, aa, bbb\}$ after padding would be $a\$\$ < aa\$ < ac\$ < bbb$, and the colexicographic order of the same strings after padding would be $\$\$a < \$aa < bbb < \$ac$.

A graph G is defined as a pair of two sets $G = (V, E)$, where $V = \{v_1, ..., v_n\}$ is the set of vertices and $E = \{e_1, ..., e_m\}$ is the set of edges, where an edge can be considered a pair of vertices $e_i = (v, u)$, where $v, u \in V$, that are connected. These edges may be directed, where they are defined as an ordered pair, in which edge $e_x = (v, u)$ that goes from v to u is distinguished from edge $e_y = (u, v)$, that goes from u to v. Edges can also have weights $w(e)$ where $e \in E$, a function that assigns a weight to each edge. Weight can represent costs and labels, but here it is important to know that it can be

a string representation of a k-mer string.

2.1 De Bruijn Graph

De Bruijn graphs are special graphs, and are especially used in computational biology [3, 20]. The de Bruijn graphs used in computational biology [20] slightly differ from the mathematical standard ones, with the ones used in computational biology being a subgraph of the mathematical standard de Bruijn graph [37]. A simple de Bruijn graph following the mathematical standard can be seen in Figure 2.1. De Bruijn graphs are defined by the order of k, and in this thesis we consider them to be created from a set of input strings $S_1, ..., S_m$, which represent sequence reads from a DNA or individual genomes. There are two ways to define a de Bruijn graph, a node-centric and an edge-centric definition.

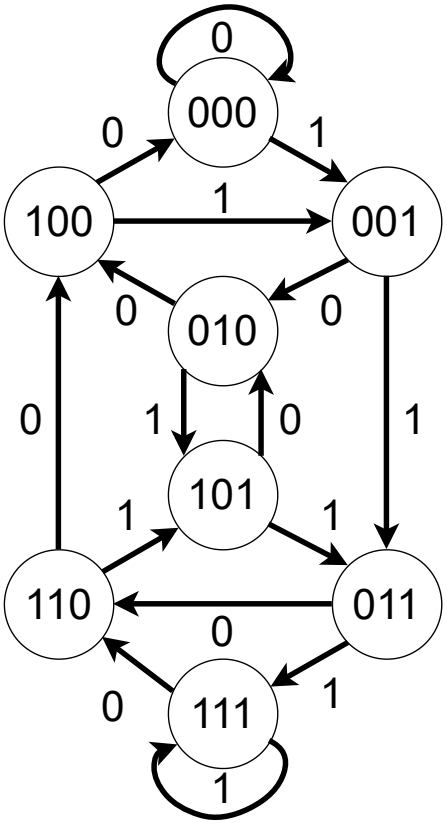


Figure 2.1: A simple mathematical standard de Bruijn graph of order $k = 4$ with the edge-centric definition over alphabet Σ where $\{0, 1\} = \Sigma$. The mathematical standard de Bruijn graph of length k has $|\Sigma|^{(k-1)}$ vertices that are all the possible string variations over alphabet Σ length of n. With the edge-centric definition, edges exist from node v to node u if v's $(k-1)$-mers suffix length of $k-2$ equals u's $(k-1)$-mers prefix length of $k-2$. The structure of the graph is from Wikipedia*.

*https://sv.m.wikipedia.org/wiki/Fil:De_bruijn_graph-for_binary_sequence_of_order_4.svg

In the node-centric definition, all nodes are distinct parsed k-mers, with an edge existing between nodes v and u if and only if the last $k-1$ characters of a node v are the same as the first $k-1$ characters of u. The node-centric graph is a subgraph of the mathematical order-k de Bruijn graph, where the alphabet is over the input strings, taking the nodes that correspond to all $|\Sigma|^k$, where Σ is the alphabet, possible input strings and adding edges between them.

The edge-centric definition has the distinct $(k-1)$-mers of the input string as nodes, with an edge existing between two nodes v and u if and only if a k-mer exists in an input string that is prefixed by v and suffixed by u. This is a subgraph of the de Bruijn graph of order $k-1$, where the alphabet is over the input strings. In the edge-centric definition, not all edges over nodes are necessarily included. This is due to the fact that even though a case occurs where node v and u have overlapping suffix and prefix length of $k-2$, the edge between v and u will not exist in the graph if the k-mer does not exist in the input. For the remainder of the thesis, the edge-centric definition will be used when discussing de Bruijn graphs.

A colored de Bruijn graph [3] is a special variation of the de Bruijn graph, where the additional information of cases a k-mer has appeared is stored. In this variation, each node has an additional label called color set. The different cases of coloring vary depending on the application. For example, a unique color may represent a single input file, or type of the sequence. A node can have multiple colors, each one signaling that this k-mer appeared in the corresponding case. As k-mers can have multiple colors, they are labeled with the set of colors they have. These color sets can consist of any possible colors. This will be further explained in Section 2.5.

The base De Bruijn graph as a concept is a crucial part of Themisto, as Themisto is a succinct de Bruijn graph, and therefore relies on its base concepts.

2.2 Succinct Data Structures

Succinct data structures are data structures that use low amount of space, but still allow efficient querying. Succinct data structures encode data efficiently, and no decoding needs to be used to query them. Every operation reads bits from the data structure, meaning that they are encoded by using 0s and 1s.

Let $T = t[0]t[1]...t[n-1]$, where $t[i] \in \Sigma$, be a string of length n and let the alphabet size be $\sigma = |\Sigma|$. Using succinct data structures, string T can be stored in $N\lceil \log_2 \sigma \rceil$ bits, and any character $t[i]$ can be retrieved in constant time using bit operations.

A common use for a succinct data structure is to compute *rank* and *select* values on strings. The rank query returns the amount of i's present in a half-open interval

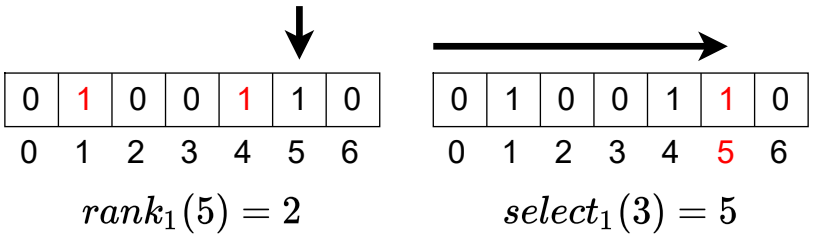


Figure 2.2: A simple representation of the results of a *rank* and *select* query on a bit vector.

$[0, i)$, or more formally $rank_c(T, i)$, where $c \in \Sigma$ and $1 \leq i \leq n$, returns the number of c's in $t[0]...t[i-1]$. For any T and c we define $rank_c(T, 0) = 0$. The select query returns the index position of j-th c in T. More formally, $select_c(T, j) = h$ where $c \in \Sigma$ and h is the index of jth c in T, where the indexing of T starts from 0. For any t and c, $select_c(T, 0) = 0$ and $select_c(T, j) = n$ for any $j > rank_c(T, n)$.

The *rank* and *select* algorithms are used in Wheeler graphs and by extension succinct de Bruijn graphs. The usage of these will be elaborated in Section 2.4.

While *rank* and *select* queries are not directly related to the Spark Themisto pipeline explored in this thesis, it is important to understand them in order to know how the graphs created by Spark Themisto are used and how they are used in the original Themisto.

2.3 Wheeler Graphs

Wheeler graphs are directed edge-labeled graphs, where nodes can be sorted by the incoming path labels [2, 15]. They can be used to explain many succinct data structures, such as succinct de Bruijn graphs created by Themisto. Wheeler graphs allow succinct storage of a graph, while allowing optimal querying. They make use of a generalized kind of Burrows-Wheeler transform [8], an algorithm that rearranges a string into runs of similar characters for the purpose of compression.

Wheeler graphs are defined as a generalization of colexicographic order of strings. The nodes of a Wheeler graph have to be able to be ordered by an order that satisfies three conditions in order to for a graph to be a Wheeler graph. Let $\lambda(e)$ symbolize the single character label of an edge that connects two nodes. The first condition is as follows: For all pairs of edges $e_x(u_x, v_x)$ and $e_y(u_y, v_y)$, if their respective labels are not equal, it means that the node with the label that has a smaller character in colexicographic order, is always smaller. Or in more formal terms:

$$1 : \text{if, } \lambda(e_x) \neq \lambda(e_y) \text{ then it follows that: } (v_x < v_y \leftrightarrow \lambda(e_x) < \lambda(e_y)).$$

The second condition states that for nodes where $u_x \neq u_y$ and $v_x \neq v_y$, for all pairs of edges $e_x(u_x, v_x)$ and $e_y(u_y, v_y)$, if the single character labels of the edges are the same, but the sources are different, the destination of the lower-source edge must not come after the destination of the higher-source edge. Formally:

$$2 : \lambda(e_x) = \lambda(e_y), \text{ then it follows that } (v_x < v_y \leftrightarrow u_x < u_y).$$

The third and final condition states that the source nodes of the graph, which are the nodes that have no incoming edges, are smaller in colexicographic order than any node with more than zero incoming edges. Note that this is true only for nodes with no incoming edges, and does not mean that nodes with less incoming edges are always smaller than nodes with more incoming edges. This can be described for a pair of nodes u, v formally as follows:

$$3 : \text{indegree}(v) = 0 \wedge \text{indegree}(u) \neq 0, \text{ then it follows that } v < u.$$

These three conditions are the Wheeler conditions that a graph has to fulfill in order for it to be considered a Wheeler graph. In short, the conditions state that the order of the nodes are decided by the incoming labels. If ties exist, they are decided by the origins of the edges, with the special case where nodes with no incoming edges coming before nodes with incoming edges.

If strings s and t are modeled in a graph as a non-branching paths, then the final nodes on the paths are ordered in a colexicographic order in the same order as colexicographic order of s and t. It is due to this why Wheeler graphs can be considered to be a generalization of colexicographic order on strings [15].

One of the features of a Wheeler graph comes from its third condition: All incoming edges to a single node have the same label, as otherwise there would be a contradiction in the form of where a node would have to be strictly smaller than itself. This also makes Wheeler graph input consistent, as all the incoming edges of a single node have the same label. Because of this, the labels can be thought to be in the nodes instead of the edges, as the edge labels can just simply be pushed to the node, with nodes that have no incoming edges having the label of ϵ, the empty character that is not in the actual alphabet and is smaller than any other character.

When a subpath query is done on a graph $G = (V, E)$, it takes a string s and returns a representation of all nodes v where a subpath exists to v with the label s. Wheeler graphs can be indexed for efficient subpath queries, which is the main motivation for its existence. The subroutines of the subpath queries used are the *rank* and *select* queries explained in Section 2.2. The detailed explanation of using these queries in Wheeler graph will be explained in the following section.

2.4 Succinct de Bruijn graphs

As mentioned in Section 1, research into more compact forms of de Bruijn graphs has risen [7]. As Themisto is a succinct de Bruijn graph, its implementation relies on the following concepts.

Bowe et al. created a succinct de Bruijn graph representation that has made it possible to represent de Bruijn graphs in a compact form. It is called the BOSS data structure [3], and it makes use of a compressed full-text substring index called FM-index [14]. FM-index in turn is based on Burrows-Wheeler transform.

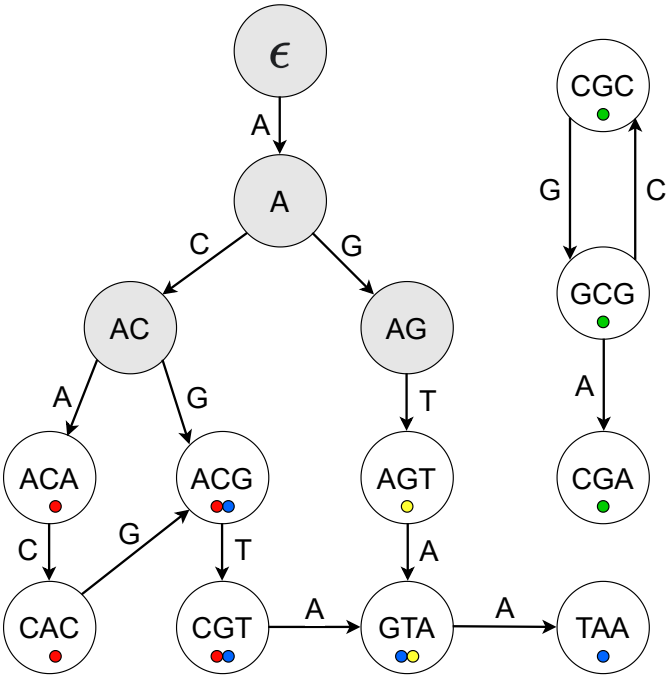
This BOSS data structure is described using strictly strings in its original paper. However, Alanko et al. represent it using a Wheeler graph index, which will be used here due to it being easier to understand.

Let W be a Wheeler graph index for a succinct de Bruijn graph G, that is created from W. The edges e of this larger graph W are labeled with single characters, and the label of an edge is denoted as $L(e)$, with the nodes having no special features and being unlabeled. As G is a de Bruijn graph, we can denote its properties using k-mers, with nodes representing $(k-1)$-mers that contain the $(k-1)$-mer label.

The label for node v is marked as $l(v)$ and they are called nodemers. The nodes V are ordered colexicographically by the labels $l(v)$ with ties broken arbitrarily. For edges e of G that represent k-mers, we denote them as $l(e)$ and call them edgemers. Edgemers include the complete k-mer, which can be thought to include the nodemer where the edge starts from, followed by the $L(e)$. Note that $l(e)$, the edgemer, contains the full k-mer, and is not to be confused with the label of an edge of G which is denoted as $L(e)$ that contains only the last character of the corresponding edgemer $l(e)$.

Due to how de Bruijn graphs work, all incoming paths to a node v that are the length of $(k-1)$ will form the same string. For example, if we have $(k-1)$-mer with a nodemer "GTA" with arbitrary amount of incoming edges, no matter which edge we will choose to traverse back, the last three edges will be "A", "T" and "G", forming the k-mer "GTA".

Because some nodes in graph G do not have $(k-1)$ length path leading to them, *dummy nodes* and *dummy edges* are added to the graph. For every node v that has indegree of 0, a path of k dummy nodes are added that lead to them, these dummy nodes representing the all possible prefixes of v, until the empty string ϵ is reached. For example, a node that has no incoming edges and has the label "AGT" and is a $(k-1)$-mer with $k=4$, will have added path of incoming edges from nodes "AG", "A" and "ϵ". The added edges that come from this have the last character of the node they are leading to as their label, so for example, in the previous example, the outgoing edge from node ϵ has "A" as its label, as it leads to node "A".



$$\text{EBWT} = A|CG|\ \ |C|\ \ |A|AG|G|G|T|T|AC|A|A$$
$$O = 10100110110100101010101001010$$
$$I = 11010101010010101010100101010$$
$$C = \{0, 6, 9, 13\}$$
$$Colors = 00216501603643, \text{ where}$$
$$0 = \emptyset, 1 = \bullet, 2 = \bullet, 3 = \bullet\bullet, 4 = \bullet, 5 = \bullet\bullet, 6 = \bullet$$

Figure 2.3: A Wheeler Graph that corresponds to a succinct colored de Bruijn graph, built from sequences "ACGTAA", "ACACGT", "AGTA", "GCGCGCGA". The indexes $EBWT$, O, and I all use the colexicographic order of the nodes. The colexicographic order here is ϵ, A, TAA, ACA, CGA, GTA, AC, CAC, CGC, AG, ACG, GCG, AGT, CGT. For example, for node CGT, we can look from $EBWT$ that it has one outgoing edge with label A, and from I and O that it has one outgoing and one ingoing edge. The C index represents the number of edges appearing in the graph according to their lexicographic order. The color index $Colors$ shows us the colors of the different nodes, by mapping each color set with a unique integer. The graph was inspired by a similar Wheeler graph as seen in [3].

If two different nodes get the same prefix added to the path leading to them, they are merged together. The dummy nodes and edges can be thought to form a tree, where the empty string is the root of the tree. The leaves of the tree are the proper prefixes of the actual nodes v of G that originally did not have incoming edges, and the leaves have the outgoing edges to these nodes. The notations $l(v)$ and $l(e)$ are extended to denote the prefixes represented by the dummy nodes v and dummy edges e with $L(e)$ representing the last character of $l(e)$.

Just like the definition of Wheeler graphs, Wheeler index on W represents the

colexicographical order of the nodes and edges of W. We define that for nodes u and v, u is smaller in colexicographical order than v if and only if $l(u)$ is colexicographically smaller than $l(v)$. The edges are also ordered in colexicographic order by using $l(e)$.

Wheeler index consists of three main components. The first component is a string called EBWT, that is the concatenation of all edge labels $L(e)$ ordered by the nodes of the origin of the edge. It is based on the Burrows Wheeler transformation [2], a method to compress a string. The second component is a bit vector O, that encodes the outdegrees of the nodes of W, again in colexicographic order. This is done by marking a start of a new node with 1, followed by the number of zeros corresponding to the amount of outgoing edges a node has. For example, a node that has zero outgoing edges will be encoded as just "1", while a node that that has two outgoing edges will be encoded as "100". The third component, I, is also a bit vector representing the amount of edges in the same way as O, but instead it represents the incoming edges to a node. The total length of I and O is the sum of the amount of edges and amount of nodes in the graph. Both O and I and how they represent nodes can be seen in Figure 2.3.

While these three components may seem to contain only a portion of the information the full graph W has, they can be used to create and define the full graph W using different operations. When querying the graph, the index C that is constructed from EBWT is also used. The index C is defined as $C[0, ...m, \Sigma|]$, such that $C[i]$ is the number of edges in E with a label smaller than the ith smallest symbol in σ. For example, for EBWT = "ACGATTA", the corresponding C for it would be "{0,3,4,5}".

2.4.1 Pattern Search

The *rank* and *select* queries explained in Section 2.2 are used to query information about the graph using the EBWT, I, O and C index, most notably querying whether a particular k-mer exists in the graph by querying a pattern. While Spark Themisto builds these indexes, querying them is currently not implemented in Spark Themisto. However, in the original Themisto, pattern searching is implemented.

Recall that $rank_c(i)$ returns the number of occurrences of symbol c on half-open interval $[0, i)$ and $select_c(i)$ returns the position of the ith occurrence of character c. A node's colexicographical ordering is denoted as j. These queries are done with de Bruijn graph being a Wheeler graph in mind. In the details for using *rank* and *select*, we assume that indexing of the nodes start from 0.

To find out the indegree of node j, we can use *select* on I. Using $I.select_1(j + 2) - I.select_1(j + 1) - 1$ tells how many 0's exist between the 1 representing the jth node and the 1 representing the $(j + 1)$th node, and as 0's tailing 1 represent edges, their amount tells us the indegree of the node. The outdegree of node works similarly,

but uses the O vector instead of I.

To get the labels of edges outgoing from node j, using *select* and *rank* in combination is required. As the vector EBWT has the edge labels of the originating nodes in colexicographical ordering, the vector O can be used to find the index numbers of the 0's that correspond to the edges of node j. Using these index numbers, the EBWT vector can then be sliced to get the edge labels of node j. Slicing EBWT through the indexes $\text{EBWT}[O.rank_0(O.select_1(j+1)) : O.rank_0(O.select_1(j+2))]$ gets us the string representation of edges of node j.

The labels of edges incoming to node j can be queried similarly, but instead of EBWT and O, indexes I and C are used. The index C can be thought to feature all the edges concatenated as a one string just like EBWT, but instead of being ordered based on the colexicographical order of the nodes, the string itself is lexicographically sorted. For example, $C = \{0, 6, 9, 13\}$ of Figure 2.3 would be "$AAAAAACCCGGGGTT$". The index C can also be thought to represent the incoming edges labels for I in the same way as $EBWT$ does for outgoing edges for O. Slicing $C[I.rank_0(I.select_1(j+1)) : I.rank_0(I.select_1(j+2))]$ will return the incoming labels for node j, but not directly, as the indexes returned must be decoded with EBWT in order to get the full string representations.

Recall by that the definition of Wheeler graphs, if nodes v and u have ordering where v is colexicographically smaller than u, and both of these nodes have an outgoing edge with the same character label $c \in \Sigma$ to nodes v' and u' respectively, and we assume that $v' \neq u'$, then we also know that $v' < u'$.

This implies that if we have an interval of nodes $[i, j]$ which are in order, and we follow the nodes with the same label, the resulting other contiguous interval of nodes $[i', j']$ is also in order.

When querying a pattern P from W that corresponds to, for example, edgemer $l(e)$, it is queried by one character at a time, by keeping up an interval of nodes where the current prefix resides. Using a pattern matching operation, we can take an interval and update it until the interval size is 1, meaning that $i = j$ where $[i, j]$ is the interval. When the intervals size is 1, it means that is the rank of the node with the edgemer $l(e)$ is found, and the rank of its node is $i = j$. Updating the interval tells us the parameters for the next *rank* or *select* operations, depending on the ongoing part of the pattern search.

The detailed explanation of querying pattern P using *rank* and *select* is a convoluted, and can be thought to done in five steps. These five steps are iterated for each character of the pattern to be matched. A representation of a single loop iteration of this query is shown in Figure 2.4. The iteration switches between looking up edge intervals and node intervals. The query *rank* can be though to give a nodes/edges

rank, or in other words, the colexicographical ordering of it. The algorithm is shown in Algorithm 1, where the steps are described in detail. Note that the algorithm features many subtractions and additions of 1, due to the fact that switching between indexing that starts from 0 and indexing that starts from 1 is required due to how Wheeler graphs and *rank*/*select* work.

Algorithm 1 Pattern Search

Require: k-mer to search K EBWT, O, I, C.

1: $[x, y] \leftarrow [0, n-1]$ ▷ Step 0
2: **for** character c **in** K **do**
3: $[a, b] \leftarrow [O.select_1(x+1) - x, O.select_1(y+2) - y - 2]$ ▷ Step 1
4: **if** $a < b$ **then**
5: **break** ▷ K-mer not found.
6: **end if**
7: $[e, f] \leftarrow [EBWT.rank_c(a), EBWT.rank_c(b+1)]$ ▷ Step 2
8: **if** $e = f$ **then**
9: **break** ▷ K-mer not found.
10: **end if**
11: $[g, h] \leftarrow [C[c] + e, C[c] + f - 1]$ ▷ Step 3
12: $[u, v] \leftarrow [I.select_0(g+1), I.select_0(h+1)]$ ▷ Step 4
13: $[x, y] \leftarrow [I.rank_1(u) - 1, I.rank_1(v) - 1]$ ▷ Step 5
14: **end for**
15: **return** x ▷ The Colexicographical rank for the k-mer

0. In Step 0, the first interval is initialized to go through the start of index O to its end, by choosing an interval from 0 to $n_n - 1$ where n_n is the amount of nodes, or in other words the amount of 1's in O.

1. In Step 1, we use the interval $[x, y]$ that we receive either from the previous iteration of the loop or from the initialization of the algorithm. Using two $select_1$ queries on O gives us a node interval $[a, b]$ that contains the 1's of interest that represent the nodes that may have an outgoing edge with the label c. Although not shown in the pseudocode, but if $y = n_n - 1$, then $b = n_e - 1$ where n_e is the amount of edges. In any case, if $a < b$, then we know that the queried pattern, or rather, k-mer is not present in the graph.

2. In Step 2, the node interval $[a, b]$ is used to locate the edge labels of interest that have the character c. Using two $rank_c$ queries, the edge interval $[e, f]$, that contains the next edges of interest, is found from the index EBWT.

3. In Step 3, using the C array, the interval $[g, h]$ is found. This interval corresponds to the nodes incoming edges, enabling the following of the edges.

4. In Step 4, the interval $[u, v]$ is found by using two $select_0$ queries on I, and this interval gives us the edges that are incoming to the nodes that we are interested to follow in the next interval.

5. In Step 5, the nodes that have incoming edges leading to them with the label c are found and marked in the interval $[x, y]$ used in the next iteration of the loop. If this is the last iteration of the loop and $x = y$, then x is the colexicographical rank of the k-mer searched.

Using $rank$ queries on EBWT or F or $rank/select$ queries on the I and O vectors can be done in $\mathcal{O}(\log \sigma)$ time. Locating pattern P from the Wheeler index is done in $\mathcal{O}(|P| \log \sigma)$ time.

In the original description of BOSS [7], nodes that have indegree or outdegree of zero are allowed. By making a limitation of not allowing these in the previously described form, we can mark the last outgoing and the first incoming edge to each node, and with this represent the information of the indegrees and outdegrees. This however comes at the cost of introducing the null character, symbolized by a \$, to the alphabet Σ, and by adding extra edges that have \$ as their label. This is to ensure that every single node has one outgoing and one incoming edge, but in spite of the additional character, it overall makes the representation of indegree and outdegree more compact.

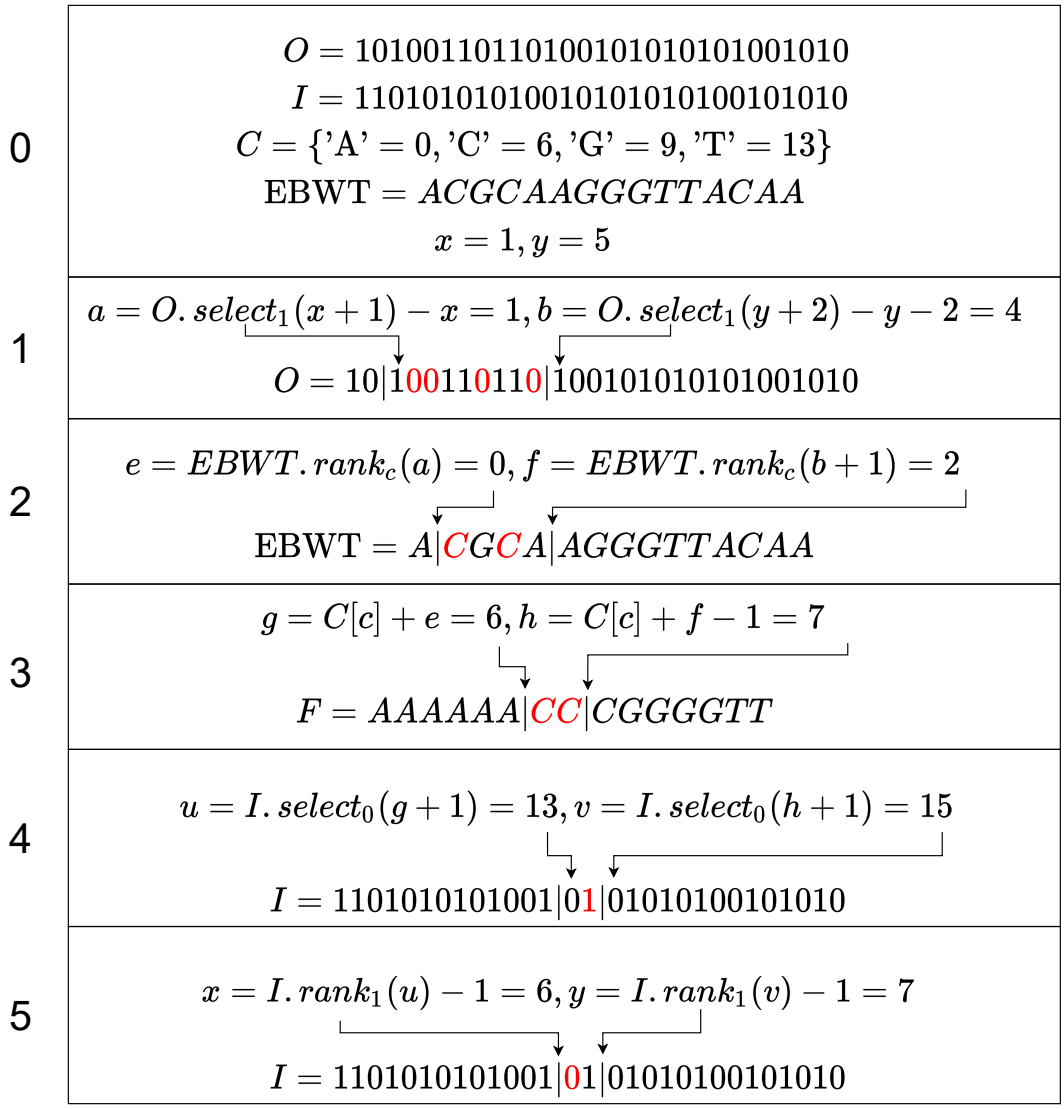


Figure 2.4: A single iteration of the k-mer pattern search loop for the graph seen in Figure 2.3. Here, k-mer ACA is searched, with the loop currently searching the character 'C' represented as c. The numbered parts correspond to the steps as seen in Algorithm 1. The iteration of the loop ends when the new x=6 and y=7 are found for the next iteration. The new x and y tell the colexicographical rank of the nodes that will be traversed upwards in the next iteration for the search of the k-mer ACA. The colexicographically ranked 6 and 7 nodes are AC and CAC, the only nodes that could have the predecessor ACA. To make the Figure easier to understand, C is represented as its string representation F in the middle of the loop.

2.5 Color Aggregation

A colored de Bruijn graph is an extension of the original where each k-mers are associated with information of its data set(s) of origin, that can be defined in various ways depending on the purpose [23]. Each data set is given a unique color, and each k-mer has a color set that defines its origin data sets.

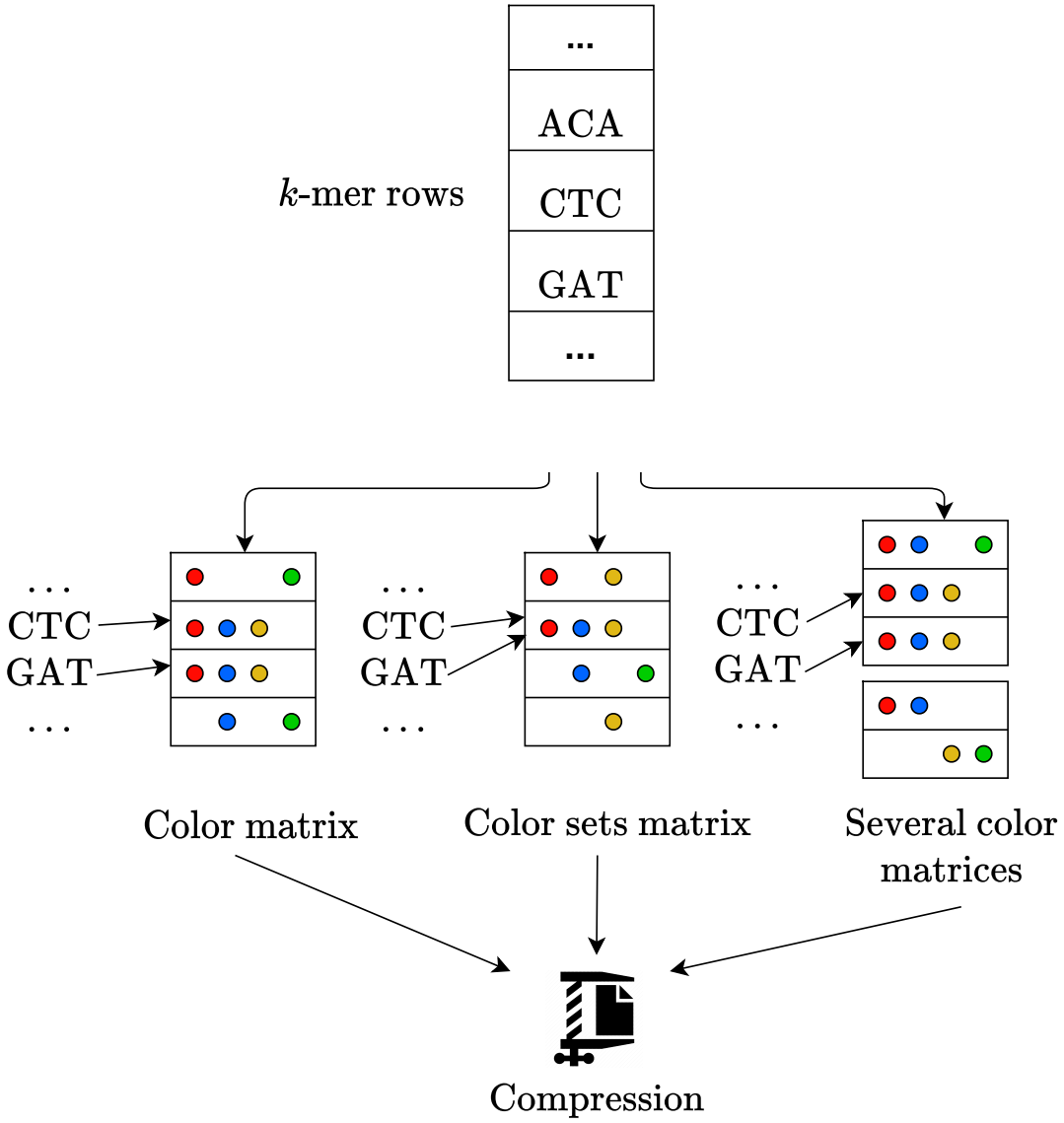


Figure 2.5: Different strategies for storing colors. From left to right, color matrix, color sets matrix and several color matrices methods. The result of any color storing strategy can be compressed using various strategies to save space.

Examples of variations of how colors can be stored can be seen in Figure 2.5. At its most basic, a bit vector can be created to represent the colors of k-mers. For example, given n k-mers and c data sets, multiple bit vectors can be combined to create a color matrix with the shape of $n \times c$ where active bit in position i, j means that color j is active in k-mer i.

A different approach is to use global color sets, where all the different combinations of colors a k-mer can have receive a unique integer that represents them. Using this method, it is possible to look up the integer that represents a k-mers color set, and then look up from another array that what color sets does the integer represent. This exploits the redundancy where many different data sets share the same k-mers,

resulting in smaller color encoding size. If two k-mers share the same color sets, they are associated with a single color set instead of two identical bit vectors.

Several matrices can also be used in conjunction by storing a probabilistic set of k-mers. However, this can lead to wrong association in terms of k-mers and colors. The details of this method is out of scope of this thesis, and the reader is referred to [23] for further explanation.

Due to the fact that many data sets share a large number of k-mers, the resulting redundancy can be taken advantage of in various different compression methods. For example, a color matrix can be compressed using bit encoding techniques, or by taking advantage of the sparsity or the density of the matrix [23]. Another example is delta-based encoding, where instead of storing two bit vectors, only the first is stored along with a list of positions that need to inverted in order to obtain the second vector.

While Spark Themisto uses color sets, no advanced methods of color compression is implemented in it. However, Spark Themisto stores only a subset of color sets as a way of avoiding redundancy by taking advantage of the properties of succinct de Bruijn graphs. This will be further explored in Section 3.1.2.

2.6 Spark

Apache Spark is a cluster-computing framework that has wide scale applications for big data processing [5]. Spark distributes computation across nodes in a cluster, by partitioning and distributing data between the nodes. It has APIs in Scala, Java and Python programming languages, and it contains libraries for streaming, graph processing and machine learning. It is open source, and makes use of an abstraction called resilient distributed datasets, or RDDs in short. Spark Themisto is implemented in Apache Spark, using the Scala API of Spark.

Spark consists of a programming model that creates a dependency graph, and an optimized runtime system which uses this graph to schedule work units on a cluster. It also transports code and data to the *worker nodes* of this cluster where they will be processed by *executor processes*. Worker nodes handle the computation, and carry a number of executors that each have their own computations. An overview of Spark Architecture can be seen in Figure 2.6.

Spark shares implementation details with earlier systems designed to handle large workloads, such as MapReduce [12] and its open source implementation Hadoop [36], but improved upon them by making optimization easier for users, and by making it possible to save intermediate results of the computation to memory [40]. MapReduce operates on input data as a set of (*key, value*) pairs, and is based on two functions, *map* and *reduce*. Map processes the inputs (*key, value*) pairs, and returns an intermediate

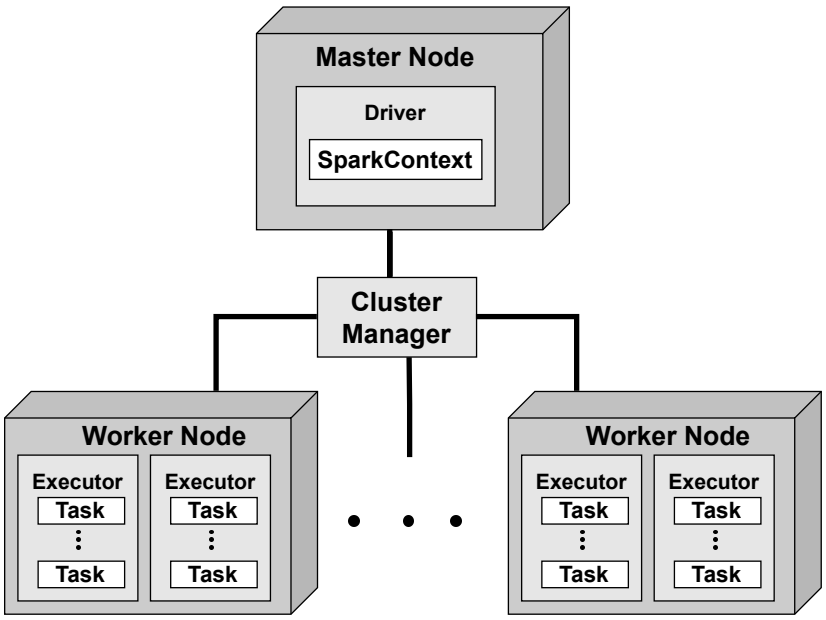


Figure 2.6: A simple overview of the Apache Spark architecture. There is a single master node that contains the driver that has SparkContext, a client which represents the connection to a Spark cluster. The master node can simultaneously act as a worker node as well. Multiple worker nodes can exist simultaneously, and each has multiple executors that execute tasks. The worker nodes do the actual computation of the Spark jobs.

set of (*key, value*) pairs. Reduce then merges the intermediate (*key, value*) value pairs to form another set, usually smaller, of values. MapReduce functions are run on nodes of a distributed computing cluster.

Apache Hadoop is a framework that supports the MapReduce paradigm, and makes use of two architectural components YARN *(Yet Another Resource Negotiator)* [38] and HDFS *(Hadoop Distributed File System)* [36]. YARN handles the resource allocation of the computing cluster, while HDFS is a distributed and block-structured file system that handles fault tolerance through data replication. Both of these are features that can be used with Spark.

MapReduce always stores intermediate results to disks, while RDDs store them in memory by default [5]. The ability to store intermediate results in memory makes more efficient data reuse between multiple computations that use intermediate results possible. This meant that there was no longer need to write the results to an external distributed file system that would incur substantial overheads due to data replication, disk I/O operations and serialization, which took a significant time of an application's total execution time.

RDDs are parallel data structures that are a read-only partitioned collection of records [40]. They can be created only by from either data that is in a stable storage, or from other RDDs. By default, Spark reads the input data into an RDD from the

nodes that are close to it. RDDs are the core feature of Spark, and enable the efficient data reuse that differentiated Spark from similar older systems.

There is no need for an RDD to be materialized at all times. This is because RDD contains the information about how it was derived from other datasets. This means that a RDD has the ability to be used to compute all its missing partitions from the data that resides in a stable storage, that it was originally created from. This information features the transformations done to the data, which are operations that are applied to many data items. This feature provides fault-tolerance, as if a partition is lost, then the information about how it was derived will be located in other RDDs and in turn can be used to recompute the lost portion. Lost data recovered this way is more efficient and less expensive than replication. This gives RDDs a powerful property, since a program cannot reference a RDD that cannot be reconstructed after a failure.

Users can also control two aspects of RDDs. First is persistence, where a user can indicate that they will reuse a RDD, making the intermediate results available for further operations. The second is partitioning, where the strategy of where a RDD is stored can be chosen, for example in memory or hard storage. RDDs can also be partitioned based on a key in each record, ensuring that two datasets that are joined together are partitioned evenly across the nodes. These partitions make it possible to optimize the transformations.

RDDs are evaluated lazily, which means that RDDs are represented as a plan of how a computation is applied to a dataset. When an action that can produce an output is reached, such as **count**, the computation is launched. This allows the Spark engine to do query optimization under the hood, meaning that some intermediate results do not need to be materialized. However, the Spark RDD engine does not understand the structure of data or the data types in RDDs, and thus it is somewhat limited and hard to make certain optimizations for.

The mechanism Spark uses to redistribute data across partitions is called *shuffling* [32]. As RDDs are evaluated lazily, shuffling only occurs when certain transformations, such as **groupBy**, are called that make the data move across different processes or between executors on separate nodes. The RDD obtained via a shuffle transformation will retain the number of partitions of the original RDD.

2.7 Spark SQL

Spark SQL is a module for Apache Spark that brings relational database query functionality to Spark, while also keeping the traditional procedural processing in the system [5]. The motivation behind creating Spark SQL was to bring relational queries that

are simpler and provide better user experience, while also still maintaining the option to implement complex procedural algorithms. The problem with pure relational systems is that they are in many ways insufficient for big data applications. Computation involving "Extract, Transform, and Load" operations, handling semi- or unstructured data and advanced analytics, such as machine learning and graph processing, are easier to express and implement in procedural computing. Spark SQL bridges the gap between these two models, allowing users to create data pipelines that are expressed in both relational queries and procedural algorithms [5].

Spark SQL provides a dataframe API, which makes it possible to perform relational queries on both external data sources and on Spark's built-in distributed collections. Dataframes are a statically typed distributed collection of rows, and can be considered to be equal to a table in a relational database. Dataframes are more efficient than the procedural API and RDDs because they are statically typed, unlike RDDs that are dynamically typed. Static typing allows Spark SQL to be optimized better, including pre-compilation of queries with known data type information. Dataframes store data in columnar format, which is significantly more compact than just Java/Python objects. It is also simpler to compute multiple aggregates in dataframes than in the traditional functional API.

Unlike RDDs, dataframes have the knowledge of the schema of the data, which enables the support of relational operations, which can be more easily optimized. This is due to the data types used by the action being known at compile time, unlike in RDDs where the data types are resolved at runtime. Dataframes can be constructed from external data sources or from existing RDDs, and they can also be considered to be a RDD of row objects, allowing RDD operations to be performed in them.

Just like RDDs, dataframes are also lazy, meaning that a dataframe object represents a logical plan to compute a dataset according to instructions. The actual execution occurs after the user calls an action that can create an output, such as saving the output to external storage. All major SQL data types are supported in dataframes, and the logical plans are analyzed eagerly, meaning that for example column names are checked that they exist in the underlying tables. Dataframes also possess the same resilient, immutability and distributed features of RDDs.

The core feature that makes efficient optimization in Spark SQL possible is the Catalyst optimizer [5], an SQL optimizer for SQL queries represented as dataframes. It adds new optimization techniques and features to address problems that are common in large volumes of data, such as handling semi-structured data and advanced analytics. It is possible to expand and customize the functions of the optimizer. Catalyst creates the logical plan of the execution. The logical plan is then optimized, for example, by combining multiple filter operations and by choosing a physical plan based on it, using

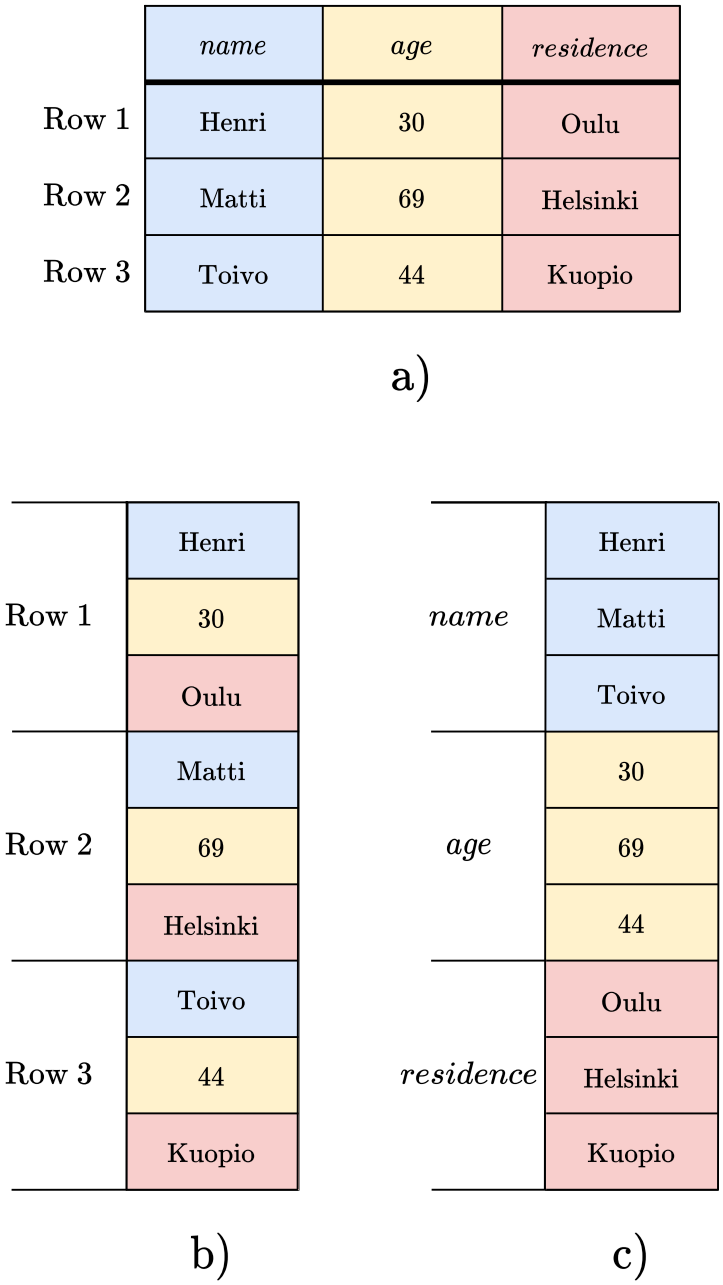


Figure 2.7: Pictured a) a simple table of data, b) how it is stored in a traditional memory buffer and c) how it is stored in a columnar memory buffer. Apache Arrow, a framework that can be used in Spark SQL, uses columnar memory buffer. Columnar memory format is used in Spark RAPIDS, introduced in Section 2.8.

the one with the lowest cost, after which the code is generated, and then executed. The plans created by Catalyst optimizer are implemented in a tree structure, with the exact details being out of scope of this work [5].

Spark Themisto uses a mix of RDD and Spark SQL processing, with the GPU-enabled version relying more on Spark SQL.

2.8 RAPIDS

As GPUs are becoming more popular in computing due to their success in deep learning, recently they have began seen use in traditional machine learning and extract transform load workloads *. While a CPU consists of a few cores that are optimized for sequential serial processing, a GPU has a massively parallel architecture that consists of thousands of smaller cores, that can handle multiple tasks simultaneously. This means GPUs are better for many computationally demanding tasks and potentially can provide massive speedups in them.

RAPIDS† is a suite of different open-source software libraries and APIs that provide GPU enabled computing to various different purposes. The motivation for these libraries is to try to provide speedup by using GPU computing. The RAPIDS libraries are built on top of NVIDIA CUDA‡, an architecture and software platform for GPU computing. The RAPIDS libraries expose the GPU parallelism through APIs, and make use of GPU dataframe objects, similar to Spark dataframes, that are stored in standardized language-independent columnar memory format. A RAPIDS library for Spark has been created for Apache Spark version 3.x, allowing Spark to use columnar processing on GPU that is much more GPU friendly than row-by-row processing. It is important to note that Spark RAPIDS is only compatible with Spark SQL, meaning that only certain dataframe operations have a GPU enabled version. RDD operations are not supported by Spark RAPIDS, and the processing switches over to a CPU if a RDD or a not supported operation occurs in a Spark program. A simple overview of the Spark RAPIDS technology stack can be seen in Figure 2.8

Spark RAPIDS an open-source project developed by NVIDIA§. Essentially acting as an extension to Catalyst optimizer, it analyzes the physical plan and replaces executor and expression nodes of the plan with GPU versions whenever possible. Spark RAPIDS makes heavy use of Apache Arrow, a Spark framework that minimizes data conversion and data serialization when the data processing pipeline includes different computing frameworks [1]. Arrow is beneficial for Spark RAPIDS due to using independent columnar memory format that is well-suited for GPU processing. The columnar memory format used by Arrow is visualized in Figure 2.8.

Spark RAPIDS tries to provide a reduction in computation time by using fewer nodes, without any changes required in the code created by the user. When RAPIDS is loaded into Spark, Catalyst query optimizer identifies which operators in a query

*https://www.nvidia.com/en-us/deep-learning-ai/solutions/data-science/apache-spark-3/ebook-sign-up/

†https://rapids.ai/

‡https://developer.nvidia.com/cuda-zone

§https://github.com/nvidia/spark-rapids

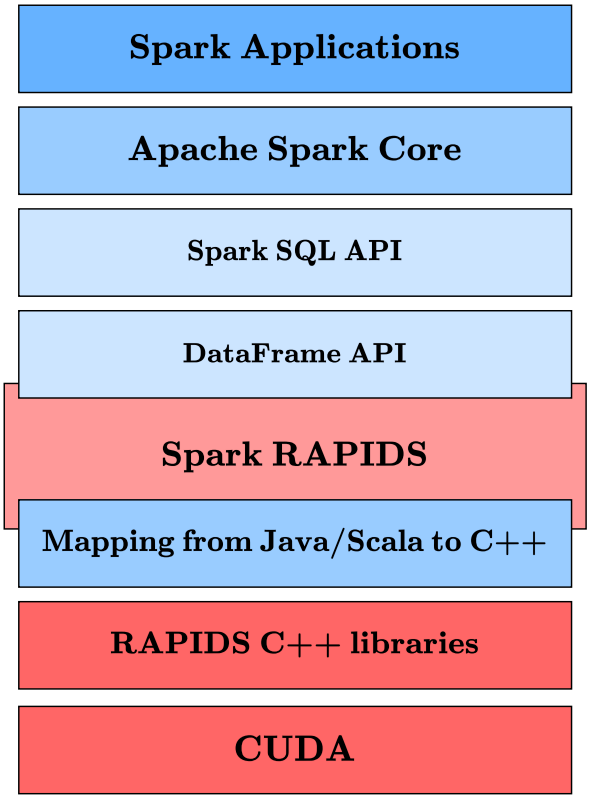


Figure 2.8: The technology stack for Spark RAPIDS. RAPIDS links to Spark SQL and dataframe API, and before runtime looks which operations are implemented in Spark RAPIDS library. These libraries use RAPIDS C++ libraries that use CUDA.

plan can be replaced with a GPU enabled one. If an operation is not implemented in RAPIDS, Spark will fall back to use the CPU version, transferring the data to CPU from GPU. Only dataframe operations are supported by Spark RAPIDS.

The library is relatively new as of writing this thesis, and is missing many core operations and features. However, it is being developed actively, with new versions released frequently. The core motivation of this thesis was to implement Spark Themisto in a GPU-enabled way by using Spark RAPIDS, and to measure the potential speedup gained in computation time when using it.

3. Methods

The goal of the thesis is to create the EBWT, I, O and color indexes of the succinct de Bruijn graph using GPU accelerated computing.

This was done by implementing a RAPIDS version of Spark Themisto. Themisto is a tool created by Alanko et al. that builds succinct colored de Bruijn graphs, that can also be used as a sequence pseudoalignment tool [22]. Spark Themisto version was created by Jaakko Vuohtoniemi, and it replicates the functionality of the original using Apache Spark and its Scala API. The original Themisto created by Alanko et al. was created in the C++ programming language. Spark Themisto implements the construction of the indexes are used to create the graph, as well as the coloring of the graph. While querying the graph for k-mers is explored in Section 2.4.1, Spark Themisto does not currently support these functionalities and thus does not support the pseudoalignment features of the original Themisto. The details of Themisto will be covered in Section 3.1.

The starting point of is as follows: We start with input that consists of sequences. The sequences are in text files and are separated with identifier lines that mark when a new sequence starts in a file. These identifier lines contain the description of the sequence succeeded by them, however the sequence specific information is not used and therefore not relevant in the actual graph building and are instead discarded. The input sequences are marked with colors during the processing. A unique color can correspond either to the start of a new identifier line, or each different file given as an input. The method used to assign colors to the sequences depends on the application. In this thesis, the colors are considered to consist of the files of origin, meaning there are as many colors as there are files in the input. The sequences itself are arbitrary length, and the size of the files they come varies, as they can be anything to a hundred gigabytes. The sequences contain letters 'A', 'C', 'G', 'T', with every other characters being filtered out.

Creation of the EBWT, I, O and color aggregation indexes of the succinct de Bruijn graph happens in two distinct phases. The details of these indexes are covered in Chapter 2. The two different phases are called Graph Building and Coloring. Graph Building constructs the indexes, while Coloring constructs the indexes that contain the

colors of the nodes.

3.1 Themisto

Themisto is a tool that implements succinct de Bruijn graph construction and a pseudoalignment algorithm for pattern searching, and is used in the mGEMS pipeline [22]. In Spark Themisto, Themisto's graph construction is divided into two distinct phases, Graph Building and Coloring, that correspond to the construction of EBWT, I, and O, and to the construction of the color aggregation indexes respectively. Graph Building consists of "KmerParser" and "KmerSort", while Coloring consists of "CoreKmers" and "ColorParser".

Themisto builds a k-mer index as a succinct de Bruijn graph, where nodes of the graph are k-mers and edges are $(k+1)$-mers. This is a variant of the BOSS structure, with each node linking to a color set with a separate data structure that stores the colors of the k-mers. Let the reference sequences be denoted as $T = T_1, T_2, ..., T_m$. Let $f_l(x)$ be the set of distinct characters that are to the left of a k-mer x of T, and $f_r(x)$ be the set of distinct characters to the right of a k-mer. The set $f_r(x)$, and for $f_l(x)$, its cardinality $|f_l(x)|$ are used in the creation of the EBWT, I and O indexes by Themisto.

Spark Themisto's index construction creates the same structure, but uses a different programming approach. In Spark Themisto, Apache Spark powered parallel computing is used, instead of the programming language C++ that was used to create the original Themisto. In the following subsections, the logic and flow of the Spark Themisto pipeline is explored.

An overview of the Spark Themisto pipeline can be seen in Figure 3.1.

3.1.1 Graph Building

Graph Building consists of two unique parts, "KmerParser" and "KmerSort". To build a succinct de Bruijn graph, a Wheeler graph is built from the reference sequences by iterating on the sets $f_l(x)$ and $f_r(x)$. This is done by first listing all the distinct $(k+2)$-mers. "KmerParser" does this by taking the raw sequences as an input. "KmerParser" outputs the unique $(k+2)$-mers that are used by "KmerSort" and "CoreKmers".

In "KmerSort", for every $(k+2)$-mer with an identical k-mer x, the cardinality of the set $|f_l(x)|$ and the set $f_r(x)$ are collected by looking at the first and last characters of the $(k+2)$-mers. The cardinality $|f_l(x)|$ contains the count of unique characters appearing on the left side of a unique k-mer, while the set $f_r(x)$ contains the unique characters appearing on the right side of a unique k-mer. This corresponds to the output seen in Figure 3.4. "KmerSort" is also responsible for creating dummy nodes,

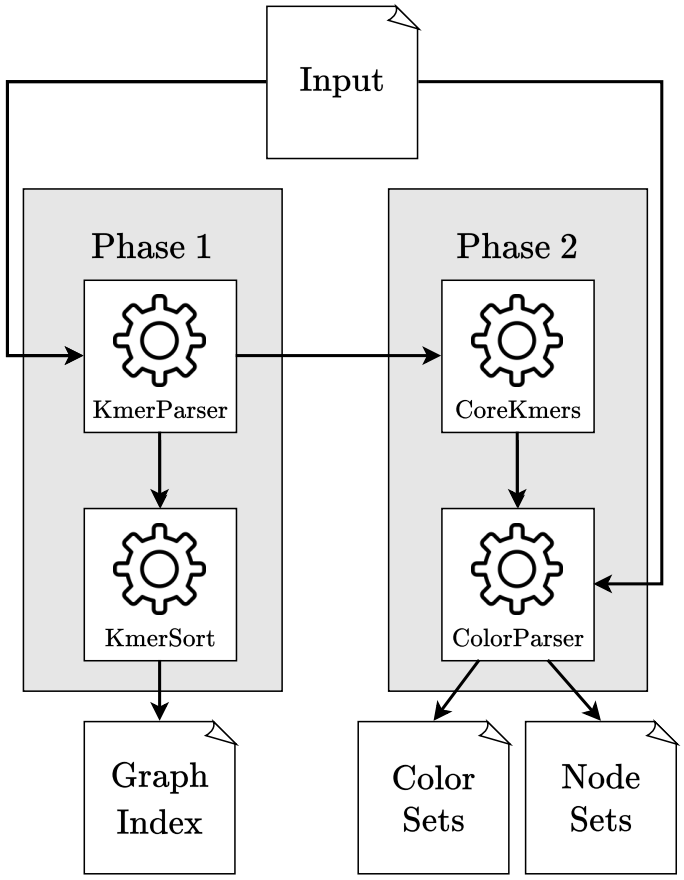


Figure 3.1: The current pipeline of Spark Themisto. The input sequences are used by both "KmerParser" and "ColorParser". "KmerParser" outputs all unique $(k+2)$-mers. Output of "KmerParser" is used in both "KmerSort" and "CoreKmers". "KmerSort" produces an output that contains all unique k-mers and their corresponding values $|f_l(x)|$ and $f_r(x)$. "CoreKmers" creates an output that contains all the core k-mers. "ColorParser" uses the core k-mers from "CoreKmers" and the original input to produce the colors for all core k-mers.

special nodes that contain the prefixes of nodes that have no predecessors, that are not considered to be k-mers.

From the information collected in "KmerSort", the Wheeler graph data structure can be created from the raw sequences $T_1, ..., T_m$. The EBWT index is created by concatenating the $f_r(y)$ of all the middle k-mer that are sorted in a colexicographic order, adding a separating symbol between them that tells when a new node starts. The O index of outgoing edges is created by making a bit vector where 1 signals the start of then next node, followed by n 0's, where the n corresponds to the amount of outgoing edges. The I index is created in the exact same way as O, but with considering incoming edges instead of outgoing. From these, the actual graph index is created, using the middle k-mers as nodes, and forming the outgoing and incoming edges from O and I. Now a working index of a succinct non-colored de Bruijn graph (V, E) is formed. This covers Graph Building, while Coloring will be covered next.

3.1.2 Coloring

Coloring consists of two unique parts, "CoreKmers" and "ColorParser". As there is a lot of redundancy due to the large amount of different colors a k-mer can have, only a subset of color sets are stored. For nodes $(u, v) \in E$, u is denoted as a predecessor of v, and v is a successor of u. The nodes that will have their color sets stored are called *core k-mers*, and are found using "CoreKmers" that takes the parsed sequences outputted by "KmerParser" as an input. The subset $V' \subseteq V$ of nodes $v \in V'$ that have their color set stored have one of the following condition hold:

1. Node v represents the first k-mer of a reference sequence.
2. The predecessors of node v represent the last k-mer of a reference sequence.
3. Node v has more than one predecessor.
4. Node v has a predecessor that has more than one successor.

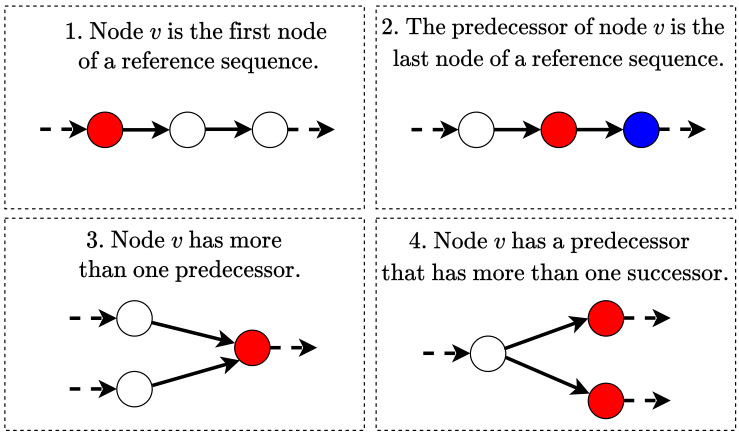


Figure 3.2: The four different cases of storing a k-mers color visualized, where a red node is considered to be a core k-mer. If a k-mer fulfills one of these cases, it is considered to be a core k-mer.

The core k-mer cases are illustrated in Figure 3.2. In the case where node $v \notin V'$, we know that node v has the same color set as its predecessor. This means that finding out its color set is done by walking backward to the nearest node $u \in V'$, which is guaranteed to exist, as the first node of every reference sequence is always in V'. "ColorParser" uses the original raw sequence data along with the output of "CoreKmers" as its input to parse color sets for the core k-mers. In the experimental runs that are referred in this thesis, the colors are assigned to the sequences based on the origin file of the sequence, although alternative options in the form of giving a file that contains the colors or giving colors based on the identifier lines are also supported.

It can take a long time to search node u for the purpose of finding out node v's color in situation of where v resides behind a long non-branching path. To make this

process faster, the original Themisto stores additional color sets for nodes in a long non-branching path. The set of nodes S have a backward distance to the nearest node in V' such that it is an integer multiple of s for some global integer parameter s. All the color sets for nodes in S are stored, making it possible to find a color set for a node at most in s backward steps. However, this is not currently implemented in the Spark Themisto.

In the original Themisto, the color sets are computed in the following way. First, we mark all nodes in $V' \cup S$, and assign reference sequences $T_1, ..., T_m$ colors such that the color of sequence T_i is i. For each $i = 0, ..., m-1$ the succinct de Bruijn graph is traversed through according to the corresponding reference sequence T_i using the constructed indexes, and if a node $v \in V' \cup U$ comes across, a pair (v, i) is stored. After processing all the reference sequences, the pairs are sorted by the k-mer identifier v. This sorted list is then scanned, and then a list containing pairs (v, C_v) is created, where C_v is the list of colors of v. Then, these pairs are sorted by the color sets and scanned through to obtain a list of pairs (X, C), where X is the set of nodes with the color set C. The pairs (X, C) are split into two different files, along with a pointer integer that combines the pair.

3.2 Code

This section covers the details of the Spark Themisto code implemented by Jaakko Vuohtoniemi and adapted for GPU computing jointly by the author. The adaptations feature changes to Spark Themisto implementation to make it more compatible with RAPIDS GPU processing. Different pieces of code feature more changes than others, with the changes being mostly replacing RDD operations with dataframe operations wherever possible to make a bigger portion of the computing time to be spent on GPU. The entire pipelines code is available in GitLab*†.

Regardless whether RAPIDS is used or not, the programs use the same input and produce the same output. However, some liberties have been taken with making the code perform better on GPU, such as using the Apache Parquet, a columnar compressed data format‡ when writing the intermediate and final outputs into disk.

Each piece of code is implemented in Scala, using a mix of Spark dataframe and RDD operations. Every time a piece of code is launched, Spark-specific operations are executed: The Spark session is created, and RDD/dataframe is formed from the input. This is a standard part of any Spark code, allowing Spark to use its libraries and show

*Graph Building https://version.helsinki.fi/xvuxvu/phase1/-/tree/Topi_Thesis_Codes
†Coloring https://version.helsinki.fi/xvuxvu/phase2/-/tree/Topi_Thesis_Codes
‡https://parquet.apache.org/documentation/latest/

various metrics in the user interface it provides. The Spark-specific operations are not included in any piece of pseudocode. Other programming-specific cases are also missing from the pseudocode descriptions. For example, in "KmerParser", a delimiter is passed to the file input reader, telling it to separate records from character ">", which marks that a new sequence starts in the file. These kinds of program specific details are hidden as they can vary in different programming languages. However, the implementation should still be straightforward in most modern programming languages.

Graph Building consists of two different pieces of code. These cover two parts in the goal of building the indexes for constructing a succinct colored de Bruijn graph. The first part, "KmerParser", parses the unique $(k+2)$-mers. The second part, "KmerSort", gives us information about the k-mers, namely the values $|f_l(x)|$ and $f_r(x)$, that are relevant when building the graph.

Coloring consists of two parts as well, that aim to collect the core k-mers and their color sets. "CoreKmers" finds the core k-mers, while "ColorParser" collects the color sets for the core k-mers. We will start by introducing "KmerParser" and "KmerSort" of Graph Building, followed by the "CoreKmers" and "ColorParser" of Coloring.

Both "KmerSort" and "CoreKmers" translated well to be run with GPUs, while "KmerParser" and "ColorParser" did not translate well to be run with GPUs. The results along with the discussion of limitations and strengths of running Spark Themisto with GPU will be covered in Sections 4 and 5 of this thesis.

The pseudocode describe the actual code as summaries of different parts. The actual code available in GitLab has comments denoting where the different portions of the pseudocodes start. The code is implemented using Scala API of Spark, with both Spark SQL and Spark RDD operations. The detailed implementation of these specific operations is not in the scope of this thesis, but documentation for them is available in the Spark dataframe manual*. We will start with Graph Building, introducing "KmerParser" followed by "KmerSort". After this, Coloring is introduced, with "CoreKmers" first and "ColorParser" after.

*https://spark.apache.org/docs/latest/api/sql/

3.2.1 KmerParser

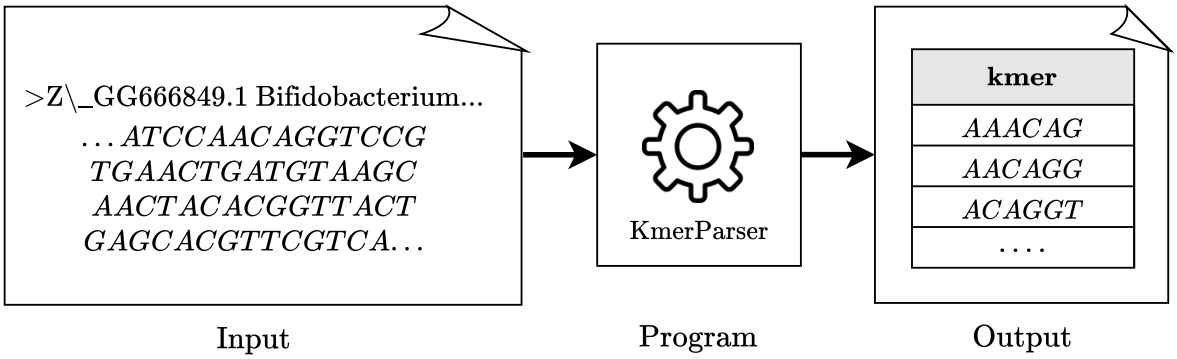


Figure 3.3: Overview of "KmerParser". "KmerParser" takes raw sequence data as input, and outputs a table of unique $(k+2)$-mers.

The first part of Graph Building is "KmerParser". "KmerParser" takes an input file that contains sequences represented as strings that are separated with an identifier line. The program outputs the parsed $(k+2)$-mers as a table that contains all the unique $(k+2)$-mers found in the input sequences. An overview of the input and output for "KmerParser" can be seen in Figure 3.3, and the pseudocode for "KmerParser" can be seen in Algorithm 2.

Algorithm 2 KmerParser

Require: List of sequences S, k-mer length to parse k

1: $S \leftarrow$ filterIllegalCharacters(S)
2: $S \leftarrow$ addNulls(S)
3: $S \leftarrow$ parseKmers(S, $k+2$)
4: **output** S

The sequence input can be thought to be a one dimensional table S, where only one column exists, and each row element contains a sequence s length of $n \in \mathbb{N}$. The table S can be thought to be a list as well. The k given as input has only the condition that $k+2$ is smaller than every sequence in the input.

The input can be thought to be a collection of sequences separated by a FASTA description lines. These FASTA description lines come from the FASTA format used to represent either nucleotide sequences or amino acid (protein) sequences, and feature a short description of the sequence that comes after the description. These description lines are not used in Themisto, and are discarded.

In line 1 of the pseudocode, the function **filterIllegalCharacters()** filters out the illegal characters and splits the sequences into a table based on the FASTA description

lines. The beginning of a new FASTA description line marks the start of a new sequence, and therefore a new row in the table.

Filtering the illegal characters in the case of Themisto means removing all characters where character $c \notin \{A, C, G, T\}$. However, removing these characters is not as straightforward as just removing them from the sequences and leaving them as they are. As illegal characters are a part of a sequence, a naive removal of them could create parsed $(k+2)$-mers in the sequence that do not exist in the actual input.

For example, a naive approach with sequence `["ACGXXAXXGTA"]` would result into a sequence of `["ACGAGTA"]`, that would result into parsed $(k+2)$-mers, where $k+2 = 3$, `["ACG", CGA", "GAG", "AGT", "GTA"]`, whereas the legitimate $(k+2)$-mers found in the original example sequence are `["ACG", "GTA"]`. The correct approach is to split and filter the illegal characters, and then filter out the non-illegal parts of the sequence that are too short to form a k-mer. For example, the sequence `["ACGXXAXXGTA"]` would result into `["ACG", "A", "GTA"]` that would, after filtering the too short sequence parts, result into sequence parts `["ACG", "GTA"]`, that are considered to be legitimate parts of the sequence to parse $(k+2)$-mers from.

In line 2, the **addNulls()** function, a null character is added to the sequences and the sequence parts beginnings and ends, to represent a sequence's start or end. In Themisto and related literature, a dollar sign \$ is often used as the null character, and is used here as well. For example, if we have sequence and parts of sequences such as `["TTGAGT", "ACG", "GTA"]`, it would result into `["$TTGAGT$", "$ACG$", "$GTA$"]`. The null characters are added as "KmerParser" parses $(k+2)$-mers, meaning that without the null characters not all k-mers and k-mers sets $f_l(x)$ and $f_r(x)$ could be collected from the $(k+2)$-mers properly in "KmerSort" and "CoreKmers".

In line 3, the $(k+2)$-mers are parsed from the sequences and their parts with **parseKmers()**. This is done using a sliding window with window size of $(k+2)$, that goes through all the sequences and sequence parts, collecting all unique $(k+2)$-mers into a table.

For example, sequence and sequence parts `["$TTGAGT$", "$ACG$", "$GTA$"]` would result into parsed $(k+2)$-mers `["$TT", "TTG", "TGA" "GAG", "AGT", "GT$" "$AC", "ACG" "CG$", "$GT", "GTA", "TA$"]`, where $(k+2) = 3$.

The unique $(k+2)$-mers are then outputted, and are ready to be used in "KmerSort" and "CoreKmers".

3.2.2 KmerSort

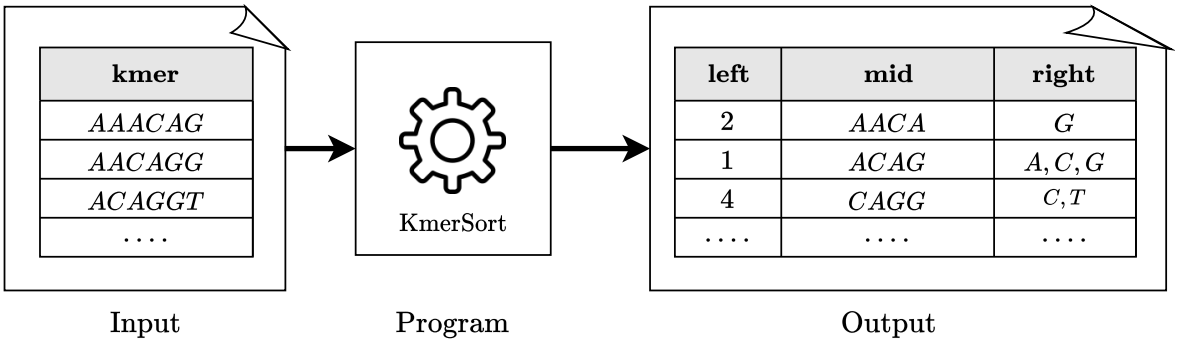


Figure 3.4: Overview of "KmerSort". "KmerSort" takes the output of "KmerParser" as an input, and outputs unique k-mers together with the amount of unique characters appearing left and the unique characters appearing right.

The second part of Graph Building is "KmerSort". The objective of "KmerSort" is to take as an input the $(k+2)$-mers parsed by "KmerParser", and use them to produce a table that has the three columns that contain information about the unique k-mers in a compact form. This information includes the unique k-mers, and the cardinality $|f_l(x)|$ and set $f_r(x)$ of each k-mer, introduced in Section 3.1.1.

The output table is formatted as follows. The middle column contains unique k-mers parsed from the inputs $(k+2)$-mers. The first column contains the integers $|f_l(x)|$ that tell how many different unique characters have appeared on the left side of a unique k-mer, and finally, the third column contains lexicographically sorted sets $f_r(x)$ that have the different unique characters that have appeared on the right side of a unique k-mer. "KmerSort" is also responsible for creating the dummy nodes explained in Section 2.4, adding them for k-mers that have no predecessors.

An overview of the input and output for "KmerSort" can be seen in Figure 3.4, and the pseudocode for "KmerSort" can be seen in Algorithm 3.

Algorithm 3 KmerSort

Require: Table of $(k+2)$-mers S

1: $S \leftarrow$ collectLeftAndRightSets(S)
2: $D \leftarrow$ where(S, $|f_l(x)| = 0$)
3: $D \leftarrow$ dummyNodeCreation(D)
4: $S \leftarrow$ when(S, $|f_l(x)| = 0$, 1)
5: $S \leftarrow S \cup D$
6: $S \leftarrow$ sortColex(S)
7: **output** S

For reference, the inputs $(k+2)$-mers are strings featuring characters "A", "C", "G", "T" and "$" appearing in a table or a list.

In line 1 of the pseudocode, the function **collectLeftAndRightSets()** collects the unique k-mers and their corresponding counts $|f_l(x)|$ and sets $f_r(x)$. For example, if we would have as an input rows $(k+2)$-mers `["ATGA", "CTGC", "GTGC", "ATG$"]`, where $k = 2$, the output would have a single row containing the following three column values `[3, "TG", ["A", "C"]]`, where the second value is the k-mer, the first the count $|f_l(x)|$, the amount of different characters found in the right of the k-mer, and the third column the set $f_r(x)$ that contains all the different characters found right of the unique k-mer. An important notion is that the null character $ does not increase the count $|f_l(x)|$ or get added to the set $f_r(x)$.

In line 2, a separate table D is created. This table will contain the dummy nodes that are created in line 3 with **dummyNodeCreation()**. For reference, dummy nodes are created due to the fact that every k-mer node must have a predecessor in a Wheeler graph. This means that only unique k-mers that have no predecessors, where $|f_l(x)| = 0$ holds for k-mer x, will be selected to have a dummy node created for them. The dummy nodes are all the unique prefixes of the k-mers that have no predecessors.

For example, for k-mers `["ACGT", "ACTT"]` with no predecessors would have the dummy nodes `["ACG", "ACT", "AC", "A", "ϵ"]` created for them, where "ϵ" is the empty node. In addition, the sets $f_r(x)$ are collected for the dummy nodes, and $|f_l(x)|$ is given as 1 for every dummy node except the empty node ϵ where $|f_l(\epsilon)| = 0$ always holds. For the previous example, this would result to table

```
[[1, "ACG", ["T"]],
 [1, "ACT", ["T"]],
[1, "AC", ["G", "T"]],
 [1, "A", ["C"]],
 [0, "ϵ", ["A"]]].
```

In line 4, the actual k-mers, that originally had no predecessors, will be marked as having a single predecessor due to the fact that they now have a dummy node as predecessor. This is done by using the **when()** function, where the second parameter marks the column and the condition to be changed, while the third parameter tells the value that the matching row/column values will be changed to. Here, we use the rows and column values where a k-mer x in table S has $|f_l(x)| = 0$ as the second parameter, and replace these with row/column values that match with the third parameter that is the integer 1.

In line 5, the two tables, one containing the original k-mers, and the other containing the dummy nodes, get merged together using a union operation.

In line 6, the table is sorted colexicographically with the function **sortColex()** based on k-mers or in other words, the middle column, after which the table is outputted.

3.2.3 CoreKmers

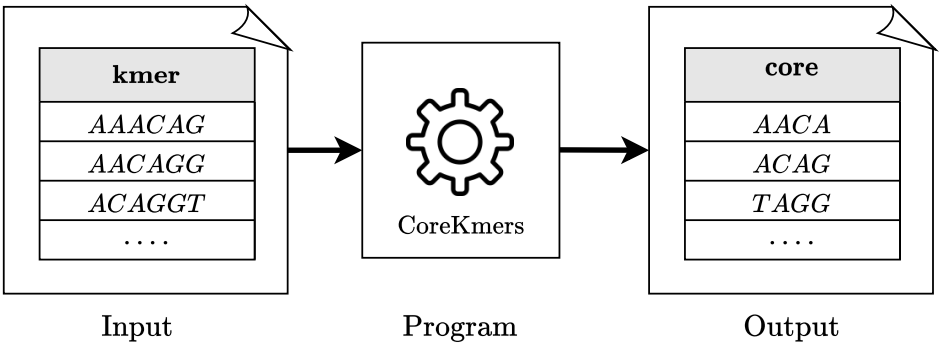


Figure 3.5: Overview of "CoreKmers". The output of "KmerParser" is taken as an input, and a table featuring the core k-mers is outputted. The different cases of core k-mers are explained in Figure 3.2.

Coloring begins with "CoreKmers", a program that takes as an input the k-mers parsed by "KmerParser", and outputs the core k-mers introduced in Section 3.1.2. The output of the program contains a table of core k-mers, the k-mers that will have their color stored. These core k-mers are then used in the second part of Coloring, "ColorParser", to collect color sets for them.

An overview of the input and output for "CoreKmers" can be seen in Figure 3.5, and the pseudocode for "CoreKmers" can be seen in Algorithm 4.

Algorithm 4 CoreKmers

Require: Table of $(k+2)$-mers S

1: $S \leftarrow$ collectLeftAndRightSets(S)
2: $S \leftarrow$ filterNonCore(S)
3: $S \leftarrow$ dropDuplicates(S)
4: **output** S

In line 1 of the pseudocode, the function **collectLeftAndRightSets()** is nearly the same as the one used in "KmerSort", but with a slight difference. The $(k+2)$-mers are split into three distinct columns that feature the unique k-mer along with the sets $f_l(x)$ and $f_r(x)$. In "CoreKmers", the set $f_l(x)$ is used instead of its cardinality, and both sets contain the null character \$ if it has appeared in the respective side of a k-mer. The null characters are required to determine if a k-mer is considered to be a core k-mer.

In line 2, all non-core k-mers are filtered out in **filterNonCore()** using the four different cases of core k-mers seen in Figure 3.2.

For example, $(k + 2)$-mers (while not a realistic parse output) `["$ACG", "ATTC", "ATTA", "CGTC"]` would result into core k-mers `["AC", "TC", "TA"]`. In this example from cases of Figure 3.2, case 1 applied for k-mer `"AC"`, while case 4 applied to k-mers `"TC"` and `"TA"`.

Afterwards in line 3, only the k-mer column is selected for outputting, and the duplicate k-mers are removed with **dropDuplicates()**, and the core k-mers are then outputted.

3.2.4 ColorParser

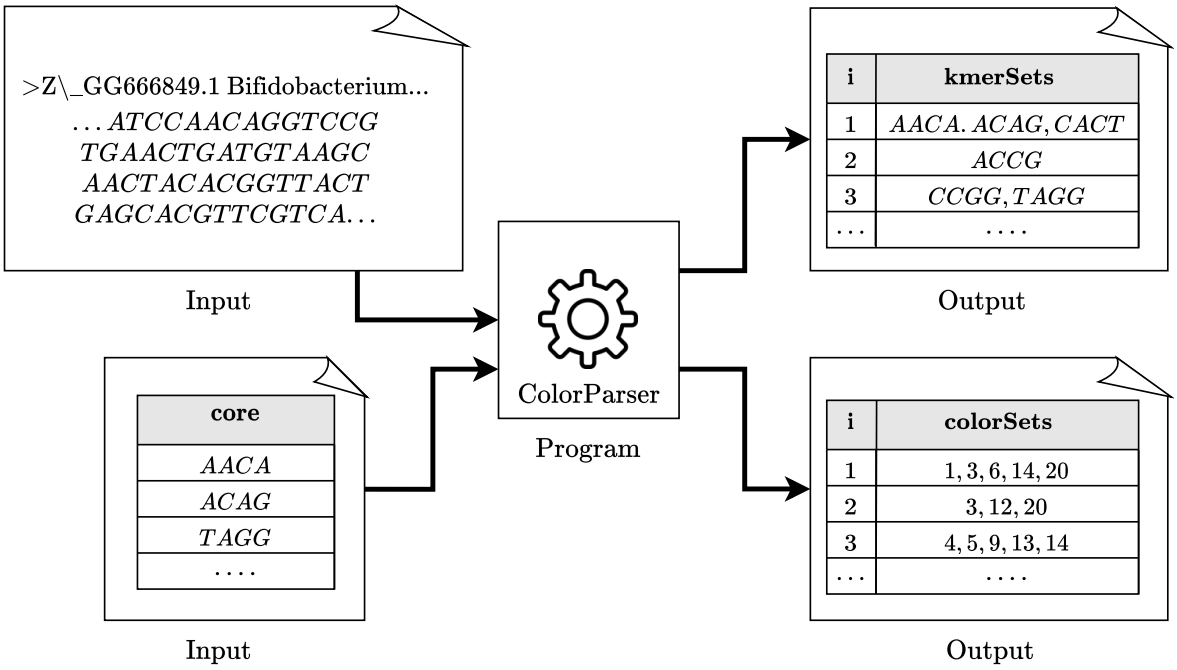


Figure 3.6: Overview of "ColorParser". The original raw sequence data and the core k-mers are taken as input, while the k-mer sets and their corresponding color sets are outputted as two separate tables.

"ColorParser" gathers all color sets for unique core k-mers, using the same original raw sequence data "KmerParser" also uses and the core k-mers outputted by "CoreK-mers". By default, "ColorParser" uses unique files as its sources for colors. For example, in the case where the original raw sequence data comes from four different files, a k-mer can have one to four colors. In our use, this method is used due to unique files containing the sequences we want to be considered distinct from each other. However, the actual code supports different formats for coloring, such as using a separate color file,

or by coloring by the sequence separators featured in the files. Here, we only consider the format suitable for our needs, where the colors are determined by the file of origin of a sequence.

An overview of the input and output for "ColorParser" can be seen in Figure 3.6, and the pseudocode for "ColorParser" can be seen in Algorithm 5.

Algorithm 5 ColorParser

Require: List of sequences with their colors S, k-mer length to parse k, table of Core k-mers C

1: $S \leftarrow$ gatherColors(S)
2: $S \leftarrow$ filterIllegalCharacters(S)
3: $S \leftarrow$ parseKmers(S.col("kmer"), k)
4: $S \leftarrow$ filter(S.col("kmer") **not in** C))
5: $S \leftarrow$ collectSetsAndIndex(S)
6: $S_{kmer} \leftarrow$ sortColex(S.col("kmers")).drop(S.col("colors"))
7: $S_{colors} \leftarrow$ drop(S.col("kmers"))
8: **output** S_{kmer}
9: **output** S_{colors}

In line 1, the colors are gathered for each individual sequence with **gatherColors()**. As mentioned before, What determines a sequence's color varies on the use case. As an example, two sequences `["CGTC", "CGTT"]` after coloring could become `[["CGTC", 1], ["CGTT", 2]]`.

In line 2, **filterIllegalCharacters()** is used as it was used "KmerParser" explained in Section 3.2.1, to format the raw sequence data into a suitable form, however, colors are also included here.

In line 3, **parseKmers()** is used to parse just as it was used in "KmerParser" in Section 3.2.1, but here k-mers are parsed instead of $(k+2)$-mers.

In line 4, all non-core k-mers are filtered using **filter()** operation and the core k-mers in table C.

In line 5, the **collectSetsAndIndex** is used to collect sets of k-mers and their corresponding color sets, as well as index the set pairs. For example, k-mers and colors

```
[["CGTC", 1],
 ["CGTT", 2],
 ["TGAT", 1],
 ["TGAT", 2],
 ["GTGT", 1]]
```

would become

```
[[1, ["CGTC", "GTGT"], [1]],
    [2, ["CGTT"], [2]],
   [3, ["TGAT"], [1, 2]]],
```

where the first column is the index, second the k-mer set, third the color set.

In lines 6 to 7, the table is split into two, with one missing the k-mer sets, and the other missing the color sets, with both having the indexes that connect the two. The table featuring the k-mer sets is sorted colexicographically, and after the proper columns are dropped from the tables, both are outputted.

This concludes the Section where Spark Themisto is explored, and the next section will present the results of running experimental tests on the pipeline.

4. Results

4.1 Experimental Setup

The implementation of GPU-enabled Themisto was tested in a set of experiments. Two different datasets were used, sized 100GB and 250GB. These datasets are partitions of the 661,405 bacterial genomes 1.5TB dataset retrieved from the European Nucleotide Archive [6]. The experiments were run with $k = 30$. These partition sizes are large enough to test the performance reliably for a Spark application, and no larger size was used due to hardware limitations.

For both of these partitions the entire Spark Themisto pipeline was tested in both CPU and GPU mode, with four different set of setup parameters with each of the different programs defined in Section 3, "KmerParser", "KmerSort", "CoreKmers" and "ColorParser", making for a total of $2 \times 2 \times 4 \times 4 = 64$ runs.

These four different setup parameters used to run the different variations of each experiment only differed in the number of executors used. As explained in Section 2.6, Apache Spark uses executors as a resource. Each executor has a fixed number of CPU cores assigned to it, in this case 10, and a set amount of RAM memory assigned depending on the amount of executors used in a run. In the case of GPU runs, each executor also has a GPU assigned to it. In the experiments, the runtimes using the setup parameters seen in Table 4.1 are collected.

GPU	CPU
2 Executors, 20 CPU Cores, 2 GPUs	4 Executors, 20 CPU Cores
4 Executors, 40 CPU Cores, 4 GPUs	8 Executors, 40 CPU Cores
6 Executors, 60 CPU cores, 8 GPUs	6 Executors, 60 CPU cores
8 Executors, 80 CPU Cores, 8 GPUs	16 Executors, 80 CPU Cores

Table 4.1: Table of the setup parameters used in experiments.

The setup used a NVIDIA DGX-1[†], equipped with 8X NVIDIA Tesla V100 32GB

[†]https://www.nvidia.com/en-us/data-center/dgx-1/

GPUs, 4X 20-Core Intel Xeon CPU E5-2698 @ 2.20GHz CPUs, 512GB DDR4 RDIMM memory, and a 10TB RAID storage. The RAID storage has 650MB/s write speed, a disk speed that might cause I/O bottleneck especially in the faster programs, but still provide enough speed for reasonably reliable results.

Apache Spark 3.2.0 and Spark RAPIDS v21.12.0 were used in running the experiments. The runtime-specific parameters used in launching the Spark applications can be found in Appendix A. A big portion of testing GPU enabled Spark Themisto went on the tuning of Spark runtime-specific parameters. These differ from the setup parameters, such as the number of executors used. The tuning of runtime-specific parameters of RAPIDS had a large impact on the runtimes, with no tuning or bad tuning resulting into extremely poor runtimes compared to an optimal tuning. There is no certainty that the runtime-specific parameters used in these experimental runs were the best ones available, but were simply found to be the best ones of all the different variations of runtime-specific parameters experimented with when running the tests on the pipeline. For the CPU runs, fewer runtime-specific parameters had to be tuned, but all the runtime-specific parameters that were not RAPIDS specific were set to the same value regardless what processing was used. A documentation of available runtime-specific parameters can be founr from*†.

For "KmerParser" and "ColorParser", the sequence files are read as .txt files, while every other intermediate and final result is read and saved in .parquet format. Note that when discussing the input sizes of 100GB and 250GB, it is referring to the size of the uncompressed raw original input sequence. The disk space used by intermediate results between the different parts of the pipeline vary greatly in size.

Due to the limitations of the Spark RAPIDS library, only rough estimates of how much processing was executed in GPU could be done. Although it is possible to see which parts of the processing can be executed in GPU, there are no feasible ways to calculate the exact time spent between CPU processing and GPU processing when running a process that makes use of both.

The results for all different parts of the pipeline will be portrayed in the same manner. The figures show the results of the GPU runtimes on the left column, and the results of CPU runtimes on the right column. The top row shows the speedup, and the bottom column show the time of the process in minutes per executor.

When comparing the results, it is important to consider the question of either expanding the processing power by adding GPUs or by adding CPUs and therefore more executors. This can be done by considering the speedups achieved with either by adding GPUs or CPU processing.

*https://spark.apache.org/docs/latest/configuration.html
†https://nvidia.github.io/spark-rapids/docs/configs.html

n_e	Parser	Sort	Core	Color	Total
2	174	19	9.6	96	299
4	90	11	5.2	54	160
6	66	8.1	3.9	44	122
8	59	7.3	3.7	41	111

(a) GPU 100GB

n_e	Parser	Sort	Core	Color	Total
2	132	66	52	150	400
4	84	41	31	90	246
6	72	36	26	84	218
8	72	34	25	78	209

(b) CPU 100GB

n_e	Parser	Sort	Core	Color	Total
2	384	32	17	450	883
4	204	17	9.7	258	489
6	144	13	7.2	258	422
8	132	12	6.8	252	403

(c) GPU 250GB

n_e	Parser	Sort	Core	Color	Total
2	330	138	102	492	1062
4	210	90	60	306	666
6	186	78	53	264	581
8	174	72	49	252	547

(d) CPU 250GB

Table 4.2: The results of the experimental runs. Columns Parser, Sort, Core and Color correspond to "KmerParser", "KmerSort", "CoreKmers" and "ColorParser" respectively, with the values reported in minutes. Four tables contain the results for the two different datasets used and GPU and CPU runs, and contain (a) GPU with 100GB, (b) CPU with 100GB, (c) GPU with 250GB and (d) CPU with 250GB.

4.2 Runtime

The individual results for each program can be seen in Table 4.2. These results indicate that adding GPU processing to Spark does increase speedup more compared to adding more CPU processing in regard to both data sizes experimented in this setup. The speedup gets bigger as more executors are added, however, this speedup is not major. The speedup differs in different programs: "KmerSort" and "CoreKmers" perform well with GPU processing, while "KmerParser" and "ColorParser" do not benefit from it as much. GPUs also perform better overall with the smaller dataset, possibly due to the lower amount of spilling runtime intermediate data on disk.

The total summed up runtime of the pipeline can be seen in Figure 4.1. The total speedup of using GPUs ends up being around half times faster than using CPU when comparing the different setup parameters. The overall speedup difference is not that large, due to the benefits of using GPUs ending up being largely lost in the slower processing time of "KmerParser" and "ColorParser". However, a completely RAPIDS enabled pipeline is faster than the regular one, even when the speedups are not impressive in these specific parts of the pipeline.

Each different part of the pipeline will be covered next that will feature graphs corresponding to the specific part of the pipeline. These include brief descriptions of

how well the specific part adapted to GPU, as well as some program specific observations related to the performance in regard to speedup, the size of data used, and amount of executors used.

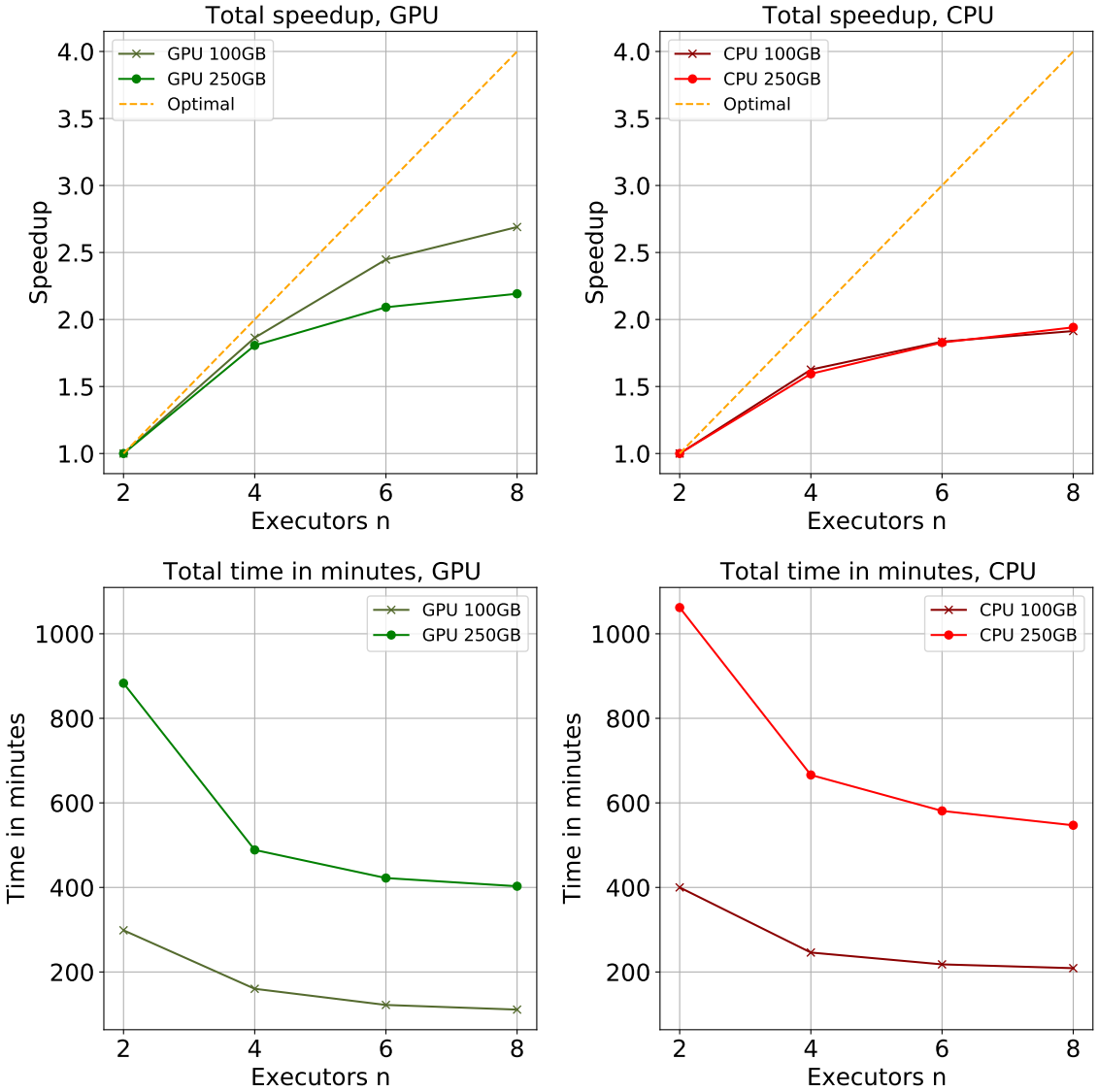


Figure 4.1: Experimental results of total runtime of the pipeline.

4.2.1 KmerParser Results

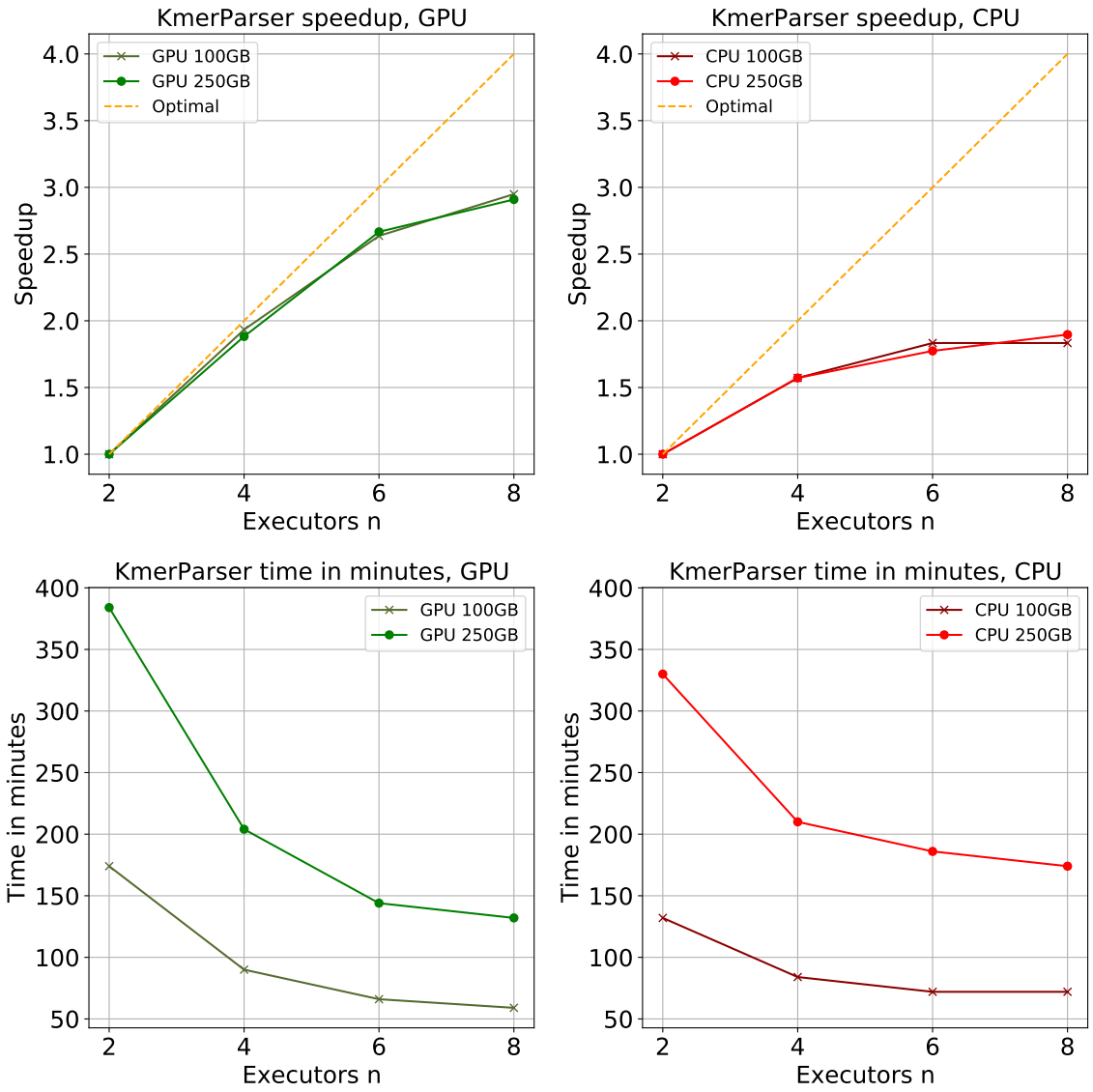


Figure 4.2: Experimental results of "KmerParser".

The results of "KmerParser" can be seen in Figure 4.2. "KmerParser" was the least suitable part of the pipeline for a RAPIDS version, as "KmerParser" uses processing that is much more feasible to do in a way that does not include Spark SQL processing. "KmerParser" implementation uses RDD operations for the most part. As Spark RAPIDS can only use Spark SQL libraries, the bulk of the runtime is executed in CPU regardless if GPU processing is enabled. Experimental setups with code that could be processed entirely in GPU proved to be ineffective and slow compared to the RDD processing.

However, the speedup from GPUs is higher than in CPUs, but this may be due

to the slow initial GPU runs with 2 executors, as it can skew the speedup of additional executors in GPUs favor. GPU processing does get faster than CPU processing as more executors are added. When running with 250GB data and 8 executors in GPU only mode, the runtime is 1.3× faster than the corresponding CPU runtime. The overall speedup is mediocre with both dataset sizes, but with 6 to 8 executors, GPU processing becomes faster regardless of the datasize used. This is even though it is costly in terms of computation time to swap the processing and data between GPU and CPU.

Overall, GPU processing achieved mediocre results in "KmerParser".

4.2.2 KmerSort Results

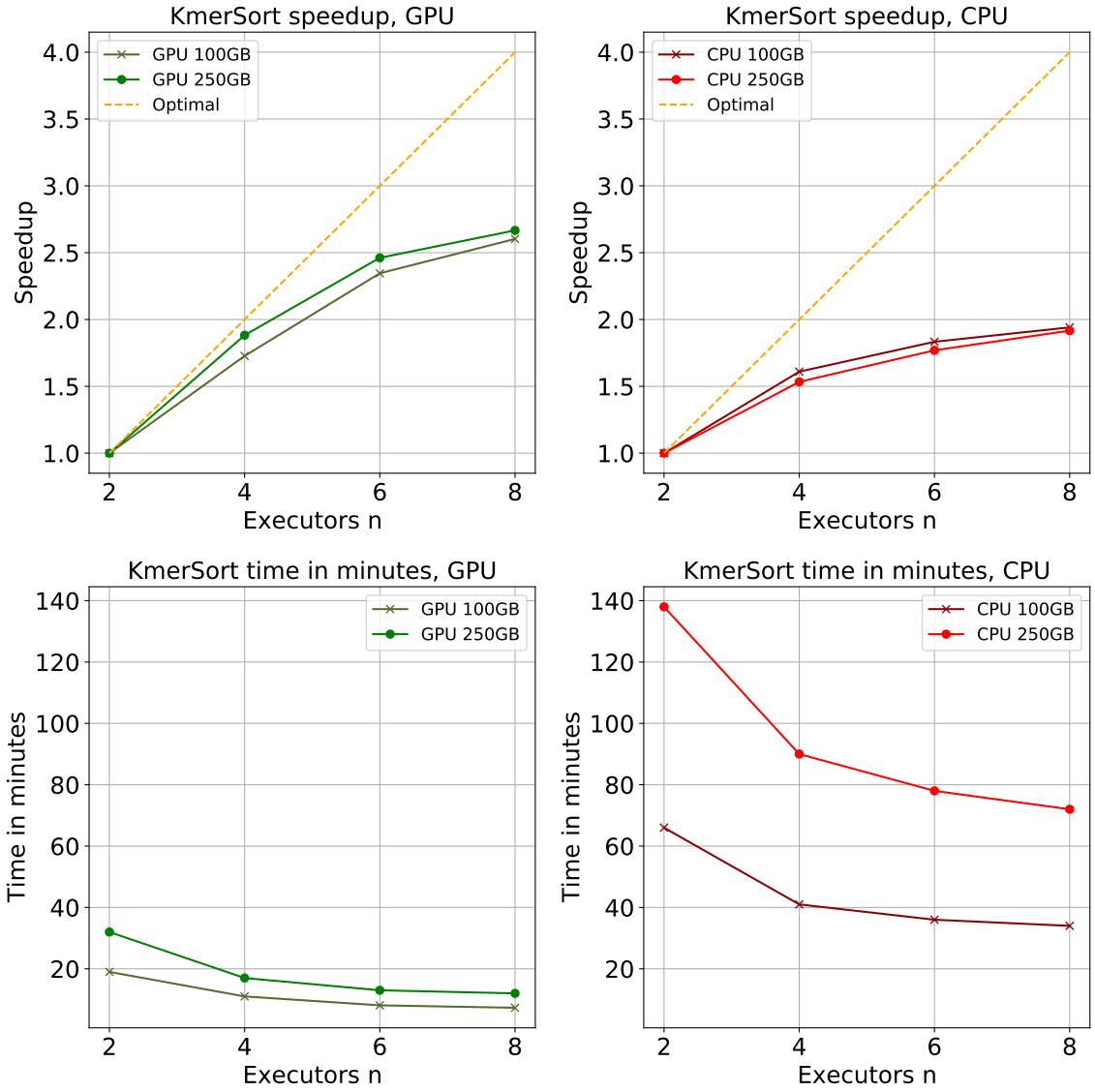


Figure 4.3: Experimental results of "KmerSort".

The results of "KmerSort" can be seen in Figure 4.3. "KmerSort" was the only part of the pipeline that was run with different code depending if run on only CPU or using GPUs. This is due to the GPU version of the code being slower than the original RDD code for CPU only processing. For GPUs, the entire processing could be done in GPU. "KmerSort" is well-suited for a dataframe implementation, making it a good fit for GPU processing.

The difference between speedups is noticeable, but because even when using only 2 executors the processing is fast, the speedup graph does not portray the true benefit of using GPUs in "KmerSort". In terms of runtime, GPU processing provides a great improvement, with runtime being over fiver times faster when running with the larger dataset and maximum processors. Reasons for this may include the low amount of swapping between the CPU and GPU processing. The speedup of GPUs in regard to computation time is best reflected when comparing the runtimes of two executors with 250GB dataset, that showcases the potential of GPUs at its best well.

Overall GPU processing proved to be effective for "KmerSort", with a large speedup between the different sized datasets and number of executors being relatively consistent.

4.2.3 CoreKmers Results

The results of "CoreKmers" can be seen in Figure 4.4. "CoreKmers", like "KmerSort", is fully compatible with GPU. "CoreKmers" was implemented in completely RAPIDS enabled way, using only dataframe operations. Of all the different programs, "CoreKmers" proved to be the most effective code to run in GPU.

For "CoreKmers", the speedup improvement when using GPUs was large, with the GPU runtime being up to seven times faster than the CPU one. "CoreKmers" proved to be effective in running GPUs even with a small number of executors, with the slowest GPU run being over two times faster than the fastest CPU run in the case of 250GB data.

In terms of executors, the speedups end up getting smaller as more executors are added, notably between 4 and 6 executors. This may be due to the data size being too small to provide an optimal speedup curve. For both sizes of the dataset used, the speedup is large and in similar scale.

Again, as with "KmerSort", the speedup achieved with GPUs may be due to the fact that no swapping between the different processing had to made, and "CoreKmers" being implemented by only using efficient and straightforward dataframe operations.

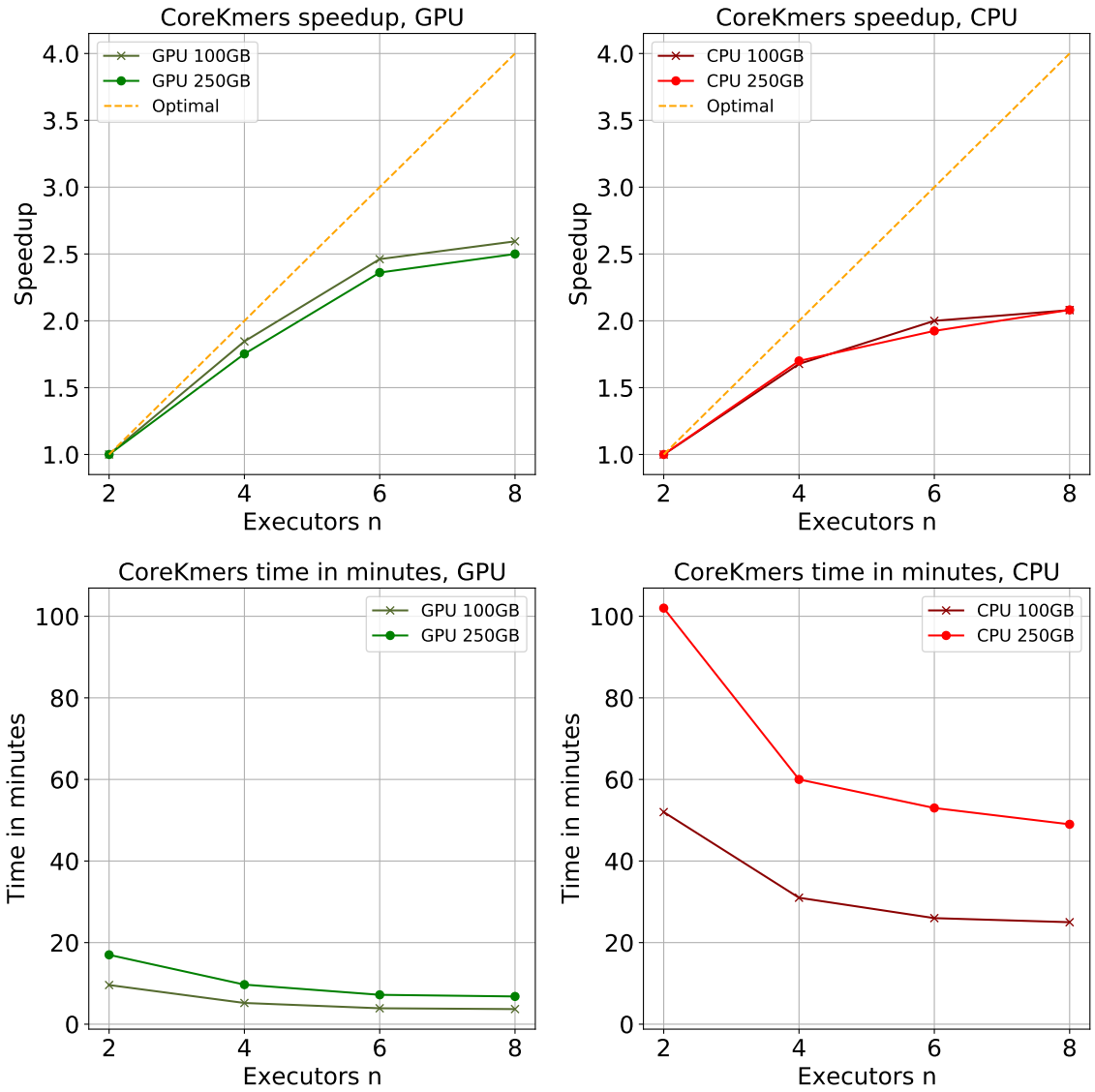


Figure 4.4: Experimental results of "CoreKmers".

4.2.4 ColorParser Results

The results of "ColorParser" can be seen in Figure 4.5. The same things that applied to "KmerParser" applies to "ColorParser" as well. "ColorParser", just like "KmerParser", was not fully compatible with GPU as it heavily relies on RDD operations that can not be run in GPU. "ColorParser" might also experience I/O bottlenecks, as both the CPU and GPU versions get hardly any speedup after 4 executors are added.

For "ColorParser", the speedup improvement when using GPUs was negligible. For 2 to 6 executors and 250GB dataset, only minor speedups are observed, while for 8 executors and 250GB dataset, no speedup is observed at all.

With the 100GB dataset, the runtime of the program is faster in GPU runs,

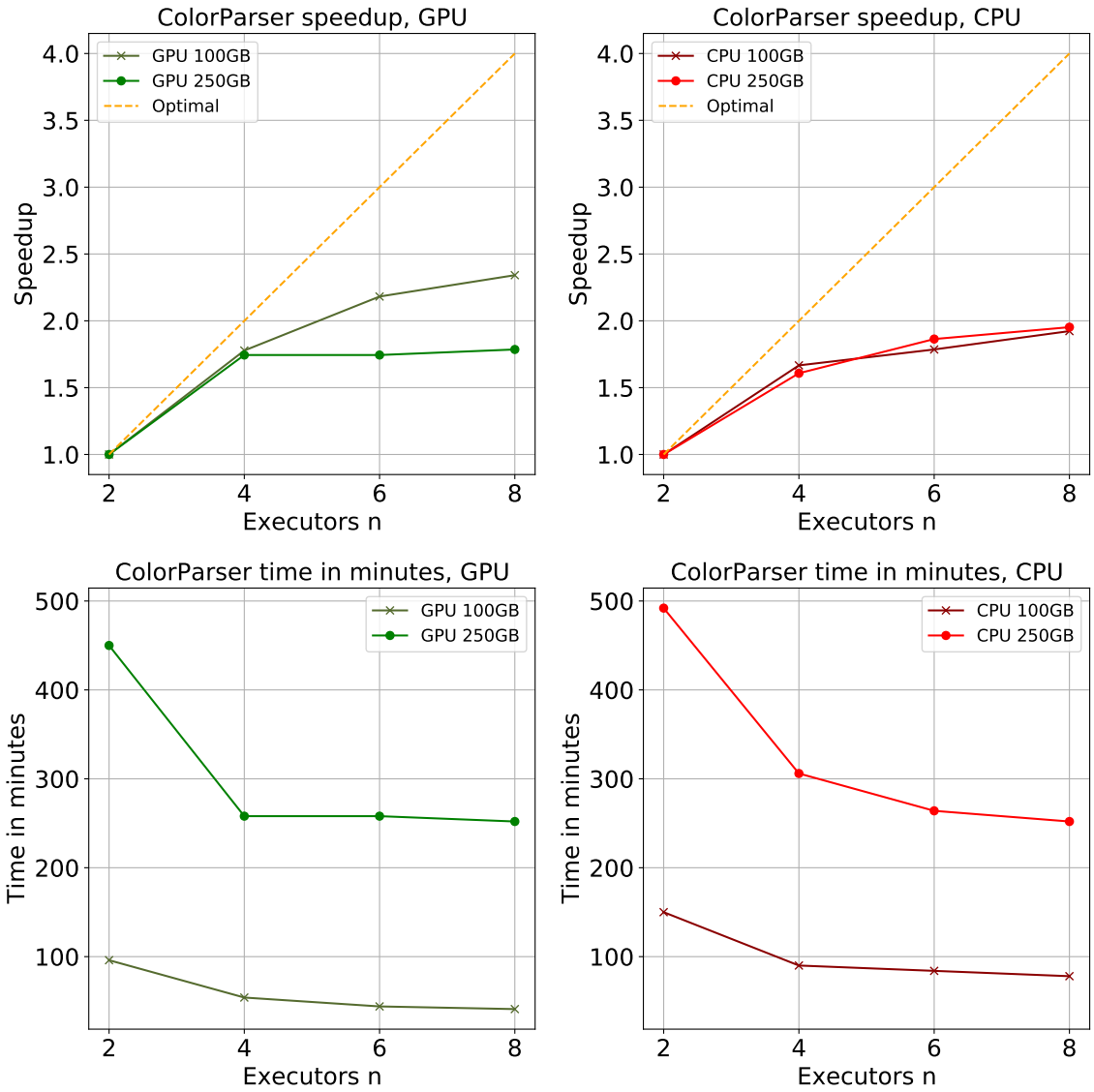


Figure 4.5: Experimental results of "ColorParser".

although the magnitude is not as great as in "KmerSort" and "CoreKmers", and is more similar to "KmerParser". The difference in runtime between CPU and GPU does get wider as the data gets bigger, but the difference is not major. The slow performance of "ColorParser" may lie in a potential I/O bottleneck, as well as in GPU runs featuring multiple swaps between the different processing types, or in an inefficient implementation.

Overall, based on the results here, GPU processing for "ColorParser" is overall not effective. When considering the further development of both Spark Themisto and GPU enabled Spark Themisto, "ColorParser" has the potential for most improvement, as it most likely bottlenecks the potential processing time of the entire Spark Themisto pipeline.

5. Conclusions

5.1 Summary

This thesis presents the Spark RAPIDS implementation of the graph building and coloring portion of Themisto, a tool that builds succinct colored de Bruijn graphs and supports pseudoalignment built by Alanko et al. [22]. This implementation uses code created for Apache Spark by Jaakko Vuohtoniemi and expands on it by modifying it for a better GPU support. The Spark RAPIDS library is a recent library, that adds GPU processing to Apache Spark, a clustering engine for large-scale data processing.

The experimental tests were gathered using a data of real genome sequence data, on two different sizes that correspond well to sizes used in actual applications. The results indicate that adding GPU processing can provide vast speedups compared to regular CPU processing when the conditions are ideal.

In the experimental results, a speedup of as big as 8× were observed. The Spark RAPIDS library is also built seamlessly on top of Apache Spark, meaning that Spark RAPIDS does not introduce its own operations or logic that affects user programming of Apache Spark, and does not require changes to existing code to be used.

5.2 Discussion

Spark RAPIDS is not perfectly suited for succinct de Bruijn graph construction, since succinct de Bruijn graphs are not easily implemented in relational processing. Spark RAPIDS is suited better for high-level and domain specific operations, while RDD processing that is not available to Spark RAPIDS offers low-level functionality and control, a better model for succinct de Bruijn graph construction.

While "KmerSort" and "CoreKmers" achieved large speedups, the time spent in the overall pipeline on them is small. Most of the runtime is focused on "KmerParser" and "ColorParser", the parts that did not translate well to GPU, but are crucial parts of the succinct de Bruijn graph construction of Themisto. If Themisto could be implemented with only dataframe operations, or rather in an efficient Spark RAPIDS

supported way, the total speedups achieved could potentially be massive, in the vein of "KmerSort" and "CoreKmers". At the moment, no efficient manner to implement Themisto in a pure dataframe operations supported by RAPIDS seem to exist, or at the very least, in a straightforward manner.

Many of the limitations of Spark RAPIDS are due to it being a relatively new library, others due to its overall framework. It does not support all existing Apache Spark dataframe operations, and notably there is no support for any type of RDD processing. This means that in many cases, not all processing can be done in GPU. This introduces severe overhead in terms of switching the processing and data between CPU and GPU processing, which is costly in terms of processing and ends up negating a sizeable portion of the speedups gained from GPU processing. Some crucial dataframe operations are also missing a GPU implementation in RAPIDS, meaning that some operations must be implemented in a haphazard manner, being inefficient and hard to understand.

Also, tuning the Spark runtime-specific parameters is much more crucial when using Spark RAPIDS than without using it. It adds a lot more parameters that are necessary to tune, as they have large importance in the amount of speedup gained when using the library. The failure of tuning these parameters correctly results in little to no speedup, or regular crashes due to out-of-memory errors. This adds an extra layer of exploration and experimentation to the Spark engine.

While using Spark RAPIDS does not limit the Spark operations that are available to use, extra steps are required to ensure that as much code as possible is supported by the library. Tools for this purpose are provided that clearly indicate the operations that are possible to execute in GPU by Spark RAPIDS. This information is given by the modified Catalyst query optimizer. This adds programming overhead when using the library to ensure the best possible results.

Spark RAPIDS has a limitation in the form of how many resources can be used when it is enabled. When using Spark, the best speedups are gained with many executors that have a small amount, typically 4 to 6, of CPU cores assigned to them. With Spark RAPIDS enabled, each Spark Executor can only have a single GPU as its resource, and there can be no more executors than there are GPUs. This limits its flexibility, as in a case where a cluster has only a few GPUs, but many CPU cores available as resources, the processing will be bottle-necked by the number of GPUs. Alternatively, the GPU processing must be forgone entirely to provide the Spark Cluster with the ideal amount of executors.

5.3 Future Work

Spark RAPIDS has many additional functionalities that are not explored in this thesis. One of these is GPUDirect Storage spilling*, a spillable cache that enables device buffers to spill directly to storage, meaning that GPUs can directly spill to disk instead of CPUs handling it. Another one is RAPIDS shuffle manager†, a feature which enables high-bandwidth transfers within nodes that has multiple GPUs.

In order to gauge the full potential of the Spark RAPIDS library, these features must be utilized. These functionalities can potentially increase the speedup provided by RAPIDS, and ending up bringing another layer of variables to consider when using RAPIDS.

The implementation of both Spark Themisto and GPU enabled Spark Themisto could be improved. The current implementation of the Spark Themisto pipeline has bottlenecks and redundancy that could be avoided. As RAPIDS is developed further, more efficient manners to implement the current functionalities will be introduced. Other similar systems to Themisto could also be implemented using Apache Spark, providing other means to further try succinct de Bruijn graph construction on Apache Spark.

Succinct de Bruijn graphs are quickly rising in popularity in the field of sequence pseudoalignment, and the research on efficient construction of them is ongoing [17, 3]. As the research on them continues, more applications that explore the usage of parallel processing and GPUs together to construct succinct de Bruijn graphs could appear, as both methods of processing individually have become a mainstay in processing large datasets.

Exploration and research of succinct de Bruijn graph construction on Apache Spark, GPUs or similar systems is still limited. However, as the size of data is already large and increasing in the field of genomics, it is clear that tools like Apache Spark and GPU processing are becoming more relevant in the field.

With systems and libraries that use GPUs and parallel processing being on the rise, the further development of these technologies is probable. In conclusion, Apache Spark provides a user-friendly platform to develop tools for large-scale data processing, and Spark RAPIDS provides a relatively seamless way to utilize GPU support for Spark. These two combined have a potential to create robust, flexible and fast data processing pipelines for applications like genome sequencing.

*https://nvidia.github.io/spark-rapids/docs/additional-functionality/gds-spilling.html
†https://nvidia.github.io/spark-rapids/docs/additional-functionality/rapids-shuffle.html

Bibliography

[1] A. Aguerzame, B. Pelletier, and F. Waeselynck. GPU acceleration of PySpark using RAPIDS AI. In S. Hammoudi, C. Quix, and J. Bernardino, editors, *Proceedings of the 8th International Conference on Data Science, Technology and Applications, DATA 2019, Prague, Czech Republic, July 26-28, 2019*, pages 437–442. SciTePress, 2019.

[2] J. Alanko. *Space-Efficient Algorithms for Strings and Prefix-Sortable Graphs*. PhD thesis, University of Helsinki, Finland, 2020.

[3] J. Alanko, B. Alipanahi, J. Settle, C. Boucher, and T. Gagie. Buffering updates enables efficient dynamic de Bruijn graphs. *Computational and Structural Biotechnology Journal*, 19:4067–4078, 2021.

[4] D. Altshuler, R. Durbin, G. Abecasis, D. Bentley, A. Chakravarti, A. Clark, F. Collins, F. De La Vega, P. Donnelly, M. Egholm, P. Flicek, S. Gabriel, R. Gibbs, B. Knoppers, E. Lander, H. Lehrach, E. Mardis, G. McVean, D. Nickerson, and R. Cartwright. A map of human genome variation from population-scale sequencing. *Nature*, 467:1061–1073, 10 2010.

[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in Spark. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394. ACM, 2015.

[6] G. A. Blackwell, M. Hunt, K. M. Malone, L. Lima, G. Horesh, B. T. F. Alako, N. R. Thomson, and Z. Iqbal. Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. *PLOS Biology*, 19(11):1–16, 11 2021.

[7] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de Bruijn graphs. In B. J. Raphael and J. Tang, editors, *Algorithms in Bioinformatics - 12th In-*

ternational Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings, volume 7534 of *Lecture Notes in Computer Science*, pages 225–235. Springer, 2012.

[8] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report, DIGITAL SRC RESEARCH REPORT, 1994.

[9] R. Chikhi, A. Limasset, and P. Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinform.*, 32(12):201–208, 2016.

[10] . G. P. Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68, 2015.

[11] Z. C. Dagdia, P. Avdeyev, and M. S. Bayzid. Biological computation and computational biology: Survey, challenges, and discussion. *Artif. Intell. Rev.*, 54(6):4169–4235, 2021.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] M. Erbert, S. Rechner, and M. Müller-Hannemann. Gerbil: a fast and memory-efficient k-mer counter with GPU-support. *Algorithms Mol. Biol.*, 12(1):9:1–9:12, 2017.

[14] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, jul 2005.

[15] T. Gagie, G. Manzini, and J. Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theor. Comput. Sci.*, 698:67–78, 2017.

[16] R. Han, N. Foutris, and C. Kotselidis. Demystifying crypto-mining: Analysis and optimizations of memory-hard PoW algorithms. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, Madison, WI, USA, March 24-26, 2019*, pages 22–33. IEEE, 2019.

[17] G. Holley and P. Melsted. Bifrost – highly parallel construction and indexing of colored and compacted de Bruijn graphs. *bioRxiv*, 2019.

[18] F. Hufsky, K. Lamkiewicz, A. Almeida, A. Aouacheria, C. N. Arighi, A. Bateman, J. Baumbach, N. Beerenwinkel, C. Brandt, M. Cacciabue, S. Chuguransky, O. Drechsel, R. D. Finn, A. Fritz, S. Fuchs, G. Hattab, A. Hauschild, D. Heider, M. Hoffmann, M. Hölzer, S. Hoops, L. Kaderali, I. Kalvari, M. von Kleist, R. Kmiecinski, D. Kühnert, G. Lasso, P. Libin, M. List, H. F. Löchel, M. J.

Martin, R. Martin, J. O. Matschinske, A. C. McHardy, P. Mendes, J. Mistry, V. Navratil, E. P. Nawrocki, Á. N. O'toole, N. Ontiveros-Palacios, A. I. Petrov, G. Rangel-Pineros, N. Redaschi, S. Reimering, K. Reinert, A. Reyes, L. J. Richardson, D. L. Robertson, S. Sadegh, J. B. Singer, K. Theys, C. Upton, M. Welzel, L. Williams, and M. Marz. Computational strategies to combat COVID-19: Useful tools to accelerate SARS-CoV-2 and Coronavirus research. *Briefings Bioinform.*, 22(2):642–663, 2021.

[19] R. M. Idury and M. S. Waterman. A new algorithm for DNA sequence assembly. *J. Comput. Biol.*, 2(2):291–306, 1995.

[20] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature genetics*, 44:226–32, 02 2012.

[21] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[22] T. Mäklin, T. Kallonen, J. Alanko, Ø. Samuelsen, K. Hegstad, V. Mäkinen, J. Corander, E. Heinz, and A. Honkela. Genomic epidemiology with mixed samples. *bioRxiv*, 2021.

[23] C. Marchet, C. Boucher, S. J. Puglisi, P. Medvedev, M. Salson, and R. Chikhi. Data structures based on k-mers for querying large collections of sequencing datasets. *bioRxiv*, 2020.

[24] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson. Adam: Genomics formats and processing patterns for cloud scale computing. Technical Report UCB/EECS-2013-207, EECS Department, University of California, Berkeley, Dec 2013.

[25] I. Minkin, S. K. Pham, and P. Medvedev. Twopaco: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinform.*, 33(24):4024–4032, 2017.

[26] D. W. Mount. *Bioinformatics - sequence and genome analysis (2. ed.)*. Cold Spring Harbor Laboratory Press, 2004.

[27] M. D. Muggli, A. Bowe, N. R. Noyes, P. S. Morley, K. E. Belk, R. Raymond, T. Gagie, S. J. Puglisi, and C. Boucher. Succinct colored de Bruijn graphs. *Bioinform.*, 33(20):3181–3187, 2017.

[28] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, 2010.

[29] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, and K. Heljanko. Hadoop-BAM: Directly manipulating next generation sequencing data in the cloud. *Bioinform.*, 28(6):876–877, 2012.

[30] I. Nisa, P. Pandey, M. Ellis, L. Oliker, A. Buluç, and K. A. Yelick. Distributed-memory k-mer counting on GPUs. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*, pages 527–536. IEEE, 2021.

[31] U. F. Petrillo, G. Roscigno, G. Cattaneo, and R. Giancarlo. Informational and linguistic analysis of large genomic sequence collections via efficient Hadoop cluster algorithms. *Bioinform.*, 34(11):1826–1833, 2018.

[32] U. F. Petrillo, M. Sorella, G. Cattaneo, R. Giancarlo, and S. E. Rombo. Analyzing big datasets of genomic sequences: Fast and scalable collection of k-mer statistics. *BMC Bioinform.*, 20-S(4):138:1–138:14, 2019.

[33] S. Qiu and Q. Luo. Parallelizing big de Bruijn graph construction on heterogeneous processors. In K. Lee and L. Liu, editors, *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 1431–1441. IEEE Computer Society, 2017.

[34] S. Ren, N. Ahmed, K. Bertels, and Z. Al-Ars. An efficient GPU-based de Bruijn graph construction algorithm for micro-assembly. In *18th IEEE International Conference on Bioinformatics and Bioengineering, BIBE 2018, Taichung, Taiwan, October 29-31, 2018*, pages 67–72. IEEE, 2018.

[35] A. Shoro and T. Soomro. Big data analysis: Apache Spark perspective. *Global Journal of Computer Science and Technology*, 15, 01 2015.

[36] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In M. G. Khatib, X. He, and M. Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.

[37] J. H. van Lint and R. M. Wilson. *A Course in Combinatorics*. Cambridge University Press, 2 edition, 2001.

[38] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia,

B. Reed, and E. Baldeschwieler. Apache hadoop YARN: Yet another resource negotiator. In G. M. Lohman, editor, *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 5:1–5:16. ACM, 2013.

[39] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In S. D. Gribble and D. Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012.

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In S. D. Gribble and D. Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012.

[41] T. Zekic, G. Holley, and J. Stoye. *Pan-Genome Storage and Analysis Techniques*, pages 29–53. Springer New York, New York, NY, 2018.

Appendix A. Runtime-specific parameters used for Spark jobs

A.1 CPU

driver-memory 24G
spark.executor.memory=(400/i)G
spark.executor.cores=10
spark.cores.max=(i*10)
spark.locality.wait=0s
spark.sql.files.maxPartitionBytes=16m
spark.sql.shuffle.partitions=1500

A.2 GPU

spark.executor.memory=(400/i)G
spark.executor.cores=10
spark.cores.max=(i*10)
spark.locality.wait=0s
spark.sql.files.maxPartitionBytes=16m
spark.sql.shuffle.partitions=1500
spark.rapids.sql.concurrentGpuTasks=8
spark.executor.resource.gpu.amount=1
spark.task.resource.gpu.amount=0.125
spark.rapids.memory.pinnedPool.size=4G
spark.plugins=com.nvidia.spark.SQLPlugin
spark.rapids.sql.csv.read.long.enabled=TRUE