Parallel Algorithms and Software Tools for High-Throughput Sequencing Data

by

Hamid Mohamadi

M.Sc., McMaster University, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate and Postdoctoral Studies

(Bioinformatics)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

June 2017

© Hamid Mohamadi 2017

Abstract

With growing throughput and dropping cost of High-Throughput Sequencing (HTS) technologies, there is a continued need to develop faster and more cost-effective bioinformatics solutions. However, the algorithms and computational power required to efficiently analyze HTS data have lagged considerably. In health and life sciences research organizations, *de novo* assembly and sequence alignment have become two key steps in everyday research and analysis. The *de novo* assembly process is a fundamental step in analyzing previously uncharacterized organisms and is one of the most computationally demanding problems in bioinformatics. The sequence alignment is a fundamental operation in a broad spectrum of genomics projects. In genome resequencing projects, they are often used prior to variant calling. In transcriptome resequencing, they provide information on gene expression. They are even used in *de novo* sequencing projects to help contiguate assembled sequences. As such designing efficient, scalable, and accurate solutions for *de novo* assembly and sequence alignment problems would have a wide effect in the field.

In this thesis, I present a collection of novel algorithms and software tools for the analysis of high-throughput sequencing data using efficient data structures. I also utilize the latest advances in parallel and distributed com-

puting to design and develop scalable and cost-effective algorithms on High-Performance Computing (HPC) infrastructures especially for the *de novo* assembly and sequence alignment problems. The algorithms and software solutions I develop are publicly available for free for academic use, to facilitate research at health and life sciences laboratories and other organizations worldwide.

Lay summary

Recent advances in DNA sequencing technologies have altered the scale and scope of health and life sciences. Using High-Throughput Sequencing technologies (HTS), huge amounts of data are produced, however, the algorithms and computational power required to efficiently and accurately process and analyze HTS data have lagged considerably. Building bioinformatics capacity to address this growing problem is an area of active research. The goal of my thesis is to design and develop novel algorithms for large-scale biological sequence analysis including *de novo* genome assembly and sequence alignment. I also utilize state-of-the-art parallel and distributed computing paradigms to build scalable and resource-efficient software solutions on high-performance computing infrastructures. The algorithms and software solutions I develop are publicly available for free for academic use, to facilitate research at health and life sciences laboratories and other organizations worldwide, and to advance knowledge about diseases and to improve human health through disease prevention and diagnosis.

Preface

Chapter 2 is focused on the biological sequence alignment problem, which involves identification of the possible coordinates of nucleotide or amino acid sequences along a target sequence, based on sequence similarity. Although there are some solutions for this problem, they have limitations in their accuracy, runtime and memory usage especially when the volume of input data is drastically increasing. In this chapter I address the challenges related to large-scale sequence alignment tasks. This work was presented at AGBT 2014 conference as a poster and in ISMB 2014 conference as a selected talk, and was published in the *PloS One* journal in 2015 [79]. I designed and implemented the algorithm and the related software tool, conducted all the experiments, and wrote the manuscript.

Chapter 3 outlines a new hashing algorithm for bioinformatics applications. Hashing is an essential operation in many bioinformatics applications for indexing, querying, and similarity search. My proposed method was presented in ISMB 2016 conference as a poster and in EMBL Big Data in Biology and Health 2016 conference as a selected talk, and was published in the *Bioinformatics* journal in 2016 [77].

Chapter 4 describes a novel algorithm for cardinality estimation in genomics data which is desirable for many bioinformatics applications and

their downstream analysis pipelines to predict genome sizes, measure sequencing error rates, and tune runtime parameters for analysis tools. This work was presented in EMBL Big Data in Biology and Health 2016 conference as a poster, and has been recently published in the *Bioinformatics* journal in 2017 [78].

Table of contents

Abstract ii

Lay summary iv

Preface v

Table of contents vii

List of tables xi

List of figures xiii

List of programs xv

Acknowledgements xvi

Dedication xviii

1 Background, motivation, and research goals 1

1.1 DNA sequencing 2

1.1.1 Next-Generation sequencing 3

1.2 Efficient data structures for NGS data 5

1.2.1 Bloom filter 6

1.2.2 Suffix tree and suffix array 6
1.2.3 BWT and FM-index 7
1.2.4 Contiguous seed 8
1.2.5 Spaced seed 9
1.3 Sequence alignment 10
1.3.1 BWA, BWA-SW, BWA-MEM 12
1.3.2 Bowtie, Bowtie2 14
1.3.3 Novoalign 15
1.3.4 GEM 16
1.3.5 SOAP, SOAP2, SOAP3 16
1.4 Sequence assembly 18
1.4.1 ALLPATHS 20
1.4.2 SGA 21
1.4.3 ABySS 22
1.5 Overview of research goals 24

2 DIDA: Distributed Indexing Dispatched Alignment 26

2.1 Publication note 26
2.2 Author summary 26
2.3 Introduction 27
2.4 Method 29
2.4.1 Assembly graph 30
2.4.2 Bloom filter 31
2.4.3 DIDA 32
2.4.4 Implementation 39

2.4.5 Evaluated tools 41
2.5 Results 42
2.5.1 Performance on real data 42
2.6 Discussion 51

3 ntHash: recursive nucleotide hashing 53
3.1 Publication note 53
3.2 Author summary 53
3.3 Introduction 54
3.4 Method 55
3.5 Results 57
3.6 Discussion 65

4 ntCard: a streaming algorithm for cardinality estimation in genomics data 69
4.1 Publication note 69
4.2 Author summary 70
4.3 Introduction 70
4.4 Method 72
4.4.1 Background, notations, and definitions 73
4.4.2 Estimating k-mer frequencies, f_i 74
4.4.3 Implementation details 81
4.5 Results 82
4.5.1 Experimental setup 82
4.5.2 Accuracy 83
4.5.3 Runtime and memory usage 85

4.6 Discussion 91

5 Conclusion 94

Bibliography 99

List of tables

2.1 Dataset specification. 43

2.2 Alignment time/indexing memory for all aligners on *C. elegans*. 43

2.3 Alignment time/indexing memory for all aligners on human draft assembly. 45

2.4 Alignment time/indexing memory for all aligners on hg19. . . 45

2.5 Alignment time/indexing memory for all aligners on *Picea glauca*. 45

3.1 Bloom filter evaluation for cityhash on real data. 61

3.2 Bloom filter evaluation for murmur on real data. 61

3.3 Bloom filter evaluation for xxhash on real data. 61

3.4 Bloom filter evaluation for ntHash on real data. 62

3.5 Bloom filter evaluation for cityhash on simulated data. 62

3.6 Bloom filter evaluation for murmur on simulated data. 62

3.7 Bloom filter evaluation for xxhash on simulated data. 63

3.8 Bloom filter evaluation for ntHash on simulated data. 63

4.1 Dataset specification. 83

4.2 Accuracy of algorithms in estimating F_0 and f_1 for HG004. . 84

4.3 Accuracy of algorithms in estimating F_0 and f_1 for NA19238. 84

4.4 Accuracy of algorithms in estimating F_0 and f_1 for PG29 . . 84

List of figures

1.1 Genome sequencing and assembly steps. 19

2.1 The workflow of DIDA with four partitions as an example. . . 30

2.2 Loading the Bloom filter using a sliding substring of length b. 36

2.3 Querying the Bloom filter using a substring of length b. . . . 36

2.4 Choosing the jumping size s for querying the Bloom filter. . . 37

2.5 Extra dispatched reads vs. r and φ. 38

2.6 Scalability of different aligners using DIDA for the *C. elegant* data. 44

2.7 Runtime scalability behaviour with the number of nodes. . . 46

2.8 The scalability performance of ABySS-map+DIDA for *C. elegans*. 47

2.9 Scalability of different aligners using DIDA for human assembly. 48

2.10 Scalability of different aligners using DIDA for hg19. 49

2.11 Scalability of abyss-map aligner using DIDA for *P. glauca*. . . 50

3.1 Correlation coefficient plots of cityhash. 58

3.2 Correlation coefficient plots of murmur hash. 59

3.3 Correlation coefficient plots of xxhash. 59

3.4 Correlation coefficient plots of ntHash. 60
3.5 Correlation coefficient of hash methods for different samples. 66
3.6 Runtime for hashing a 250bp DNA sequence. 67
3.7 Runtime for hashing 1 billion k-mers. 67
3.8 Runtime for canonical hashing 1 billion k-mers. 68
3.9 Comparing multi-hashing runtime. 68

4.1 64-bit hash value generated by ntHash. 75
4.2 The workflow of ntCard algorithm. 80
4.3 k-mer frequency histograms for human genome HG004. . . . 86
4.4 k-mer frequency histograms for human genome NA19238. . . 87
4.5 k-mer frequency histograms for white spruce genome PG29. . 88
4.6 Runtime of DSK, ntCard, KmerGenie, KmerStream, and Khmer for HG004 dataset. 90
4.7 Runtime of DSK, ntCard, KmerGenie, KmerStream, and Khmer for NA19238 dataset. 90
4.8 Runtime of DSK, ntCard, KmerGenie, KmerStream, and Khmer for PG29 dataset. 91

List of programs

2.1 The DIDA algorithm 40

4.1 The ntCard algorithm 79

Acknowledgements

I would like to express my enduring gratitude to my supervisor, Inanc Birol. His continuous and unconditional help and support made my Ph.D. work and studies at UBC and Genome Sciences Centre (GSC) at BC Cancer Agency exciting and joyful. I learnt a lot from Inanc during my time at GSC. Discussions with him not only shaped my research directions and career path, but also taught me many useful things at a higher schema. I feel incredibly indebted to him. I would like to sincerely thank Dr. Steven J M Jones, Dr. Anne Condon, and Dr. Alan Wagner who graciously accepted to be in my Ph.D. Supervisory Committee and helped me with valuable discussions and comments.

I have also been very fortunate to work with Clay Breshears and his group during my visits to Health and Life Sciences group at Intel Corporation in Hillsboro. Clay and his group introduced me to the next-generation computer systems, and helped me utilize them in bioinformatics applications.

I would also like to thank all my lab-mates and friends in the Bioinformatics Technology Lab at GSC for their comments and discussion during my Ph.D. studies. I also express my profound affection and appreciation to Nazanin for her continuous encouragement and support during my years of

education. Last, but not least, I heartily thank my family for the strong motivation that they gave me to follow my studies. Their support was invaluable to me.

Dedication

To my parents.

Chapter 1

Background, motivation, and research goals

Gathering large volumes of data by measuring physical processes is an essential feature of contemporary science. Experiments that gather massive amounts of data have been recently popular in computational biology, physics, chemistry, and finance. With recent advances in health and life sciences technologies, using high-throughput sequencing instruments, we can now produce terabytes of DNA sequencing data from previously unknown organisms, whole human populations, and thousands of human cancers, drastically increasing the computational burden of the related data analysis. Therefore, there is an essential demand for efficient, scalable, and cost-effective algorithms and software tools for analyzing the produced data.

The goal of my PhD research is to design and develop computationally efficient and scalable algorithms and software tools for accurately processing of raw DNA sequence data produced by new sequencing technologies. The algorithms and software tools that I develop, in their cores, utilize succinct, compact, and compressed data structures. These data structures become more efficient in memory use as the redundancy in the data increases, while

retaining the ability to perform efficient approximate matching. Using the algorithms I develop, I build a package to address two major problems in sequence analysis, sequence alignment and *de novo* assembly. The first problem that I address is biological sequence alignment, which involves identification of the possible coordinates of nucleotide or amino acid sequences along a target sequence, based on sequence similarity. Although there are some solutions for this problem, they have limitations in their accuracy, runtime and memory usage especially when the volume of input data is drastically increasing. The second problem that I tackle is the efficient reconstruction of a genome - the full complement of DNA within a cell - using only the output from the new sequencing instruments. Given the scale of these problems, often requiring hundreds of gigabytes of data, computation time and memory efficiency is a primary concern. I optimize memory usage and execution time requirements for *de novo* assembly problem.

In the remainder of this chapter, I introduce DNA sequencing technologies, give an overview of efficient data structures for next-generation sequencing data, and explain key problems of sequence alignment and *de novo* assembly. I will also review existing methods for the sequence alignment and *de novo* assembly problems. At the end of this chapter, I will present the overview of my research goals.

1.1 DNA sequencing

DNA sequencing is the process of finding the nucleotide orders within a DNA molecule. With the limitations of current sequencing technologies,

it is not possible to determine the whole sequence of a genome at once, but a small DNA fragment up to a few thousands of nucleotides can be sequenced. The process of DNA sequencing starts by first shearing multiple copies of the target genomes into very short DNA fragments. Afterward, the short DNA fragments are sequenced separately using DNA sequencing instruments. There are two categories for sequencing techniques: Sanger Sequencing and Next-Generation Sequencing (NGS). The Sanger sequencing technique is the first method of DNA sequencing developed by Frederick Sanger, and is based on the chain termination principle [92]. It works by incorporating chain-terminating dideoxynucleoside triphosphates (ddNTPs) during DNA replications using DNA polymerase. Before replaced by the Next-Generation Sequencing, Sanger sequencing was the most widely used approach for nearly quarter-century. In the Next-Generation Sequencing, the long DNA sequence is sheared into short DNA chunks and then DNA adapters are connected to the ends of chunks to prepare a short DNA chunk library [33]. The library will then be amplified before the sequencing process. Sanger sequencing is a expensive process with low throughput but approximately long reads, while NGS produces huge amounts of short reads faster at lower cost by simultaneously sequencing many DNA fragments.

1.1.1 Next-Generation sequencing

The first NGS technology developed, named “pyrosequencing”, was commercialized by Roche/454 Life Sciences and it is the technology that was used to sequence the genome of James Watson [107]. The pyrosequencing sequencing, instead of terminating the chain amplification in Sanger sequencing, is

based on detecting pyrophosphates in the process of nucleotide incorporation [70]. In this approach of sequencing, a single-stranded DNA is captured by beads, and then loaded into picoliter reaction wells after amplification. After loading and in a predefined order, fluorescently labeled nucleotides are included into the reaction. Subsequently, a small pulse of light is issued when a labeled nucleotide is bound to the template DNA and can be detected using a charge-coupled device (CCD) camera [33]. The reagents are then cleared after each reaction and before the next addition of nucleotides. Finally, the sequence of each template molecule is identified in real time by analyzing the captured images [107]. This technology of sequencing generates much more data per run than Sanger sequencing, yielding up to 700 Mbases of sequencing data with maximum 1000bp read lengths in about one-day run [95].

The second NGS technology was developed at the University of Cambridge and commercialized by Solexa that was acquired by Illumina later. In this technology, a template DNA is attached to sequences fixed on a slide. A cluster of molecules is then created by amplifying the template DNA in place. The action of sequencing happens during some cycles. In each cycle, fluorescent-dye labeled nucleotides, which are reversibly-terminated, are inserted into the reaction [95]. The process follows by taking an image and then chemically removing the dye and terminator to allow the reaction to proceed to the next base. After finishing the run, the captured images are processed and by using the color of each cluster the identity of which base was included during each cycle is identified [5]. This technology now generates over 1.5 Tbases per run in less than 72-hour run with maximum read

lengths of 150 bp taken from both ends of DNA fragments. The technology is also able to generate longer length reads (*e.g.* 250 bp) in other modes of operation.

Other NGS technologies are sequencing-by-ligation SOLiD by Applied Biosystems and sequencing based on semiconductor Ion Torrent by Life Technologies. Sequencing technologies continue to grow by emerging of single molecule real-time sequencing approaches requiring no DNA amplification (PacBio, HeliScope, Oxford Nanopore) and providing very long reads, but with high error rate and low accuracy [15, 33].

With the improvements in NGS technologies, huge amounts of data have been generated and this empowered researchers in investigating human genome variation and sequencing thousands of human cancers to find disease-causing mutations and genes associated with diseases. On the other hand, new sequencing technologies have also imposed new computational challenges for the analysis of enormous volumes of data being generated. Hence, we need efficient algorithms and software tools that can be scalable and cost-effective to handle the data produced remarkably in routine problems such as sequence alignment and *de novo* assembly.

1.2 Efficient data structures for NGS data

In this section, I bring some notations and definitions related to data structures for the analysis of next-generation sequencing data.

1.2.1 Bloom filter

A Bloom filter [9] is a probabilistic space-efficient and compact data structure that is designed to support membership queries on dynamic sets with an adjustable false positive parameter. It has been broadly employed in many computer science and engineering applications leveraging its capability to compactly describe a set and readily detect items that do not belong or may belong to the set with a controlled false positive rate. In bioinformatics and computational biology, the Bloom filter data structure has been used in applications such as genome assembly, k-mer counting, and error correction [14, 16, 35, 53, 89, 90, 101, 102].

The answer of querying a Bloom filter is: *no* which means the the queried element is not definitely present in the data structure, or *yes* which means the element *maybe* present in the filter. The uncertainty comes from the fact that different items can be mapped to same entries in the Bloom filter resulting in false positive hits with the following rate:

$$F = (1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{-k/r})^k \tag{1.1}$$

where n is the number of items in the filter, m is the number of bit positions in the filter, k is the number of hashes per item, and $r = m/n$ is the number of bits per element [9].

1.2.2 Suffix tree and suffix array

Suffix arrays and suffix trees [67, 68] are two widely-used data structures for indexing, text searching, and information retrieval. They have been

widely used in bioinformatics and computational biology applications. There are two specific types of substrings $s[i..j]$ for a given string s that are vey important. The substring $s[i..n]$ is a *suffix* of s for $i \in [1..n+1]$. The substring $s[1..j]$ is a *prefix* of s for any $j \in [0..n]$. For a given string s, the *suffix tree* is a tree whose leaves represent all the suffixes of s, where each suffix of s is shown in the tree as a path starting from the root and ending at the corresponding leaf. For a given string s, the *suffix array* is an array of integers denoting the start positions of the lexicographically sorted suffixes of s [76]. The Longest Common Prefix, or LCP, of two strings is the longest string that is the prefix of both strings. LCP is stored alongside the list of prefix indices, and represents the number of characters a particular suffix shares with the suffix exactly above it, starting at the beginning of both of them [76]. This value is useful to make string operations more efficient. For instance, LCP may be used to speed up searching in the list of suffixes by avoiding comparison of characters that are already known to be the same.

1.2.3 BWT and FM-index

Burrows-Wheeler Transform (BWT) for a given string T, B_T, is a reversible permutation of T, and it has been widely used in text compression and indexing. When BWT is coupled with some small auxiliary data structure, the two form a space-efficient and compact index of T [26]. By appending an occurrence array, Occ, and an aggregate count vector, C, Ferragina and Manzini [26] constructed the FM-index as a novel indexing structure. Let $C(a)$ be the index in SA_T of the first suffix that starts with character a. If v is the number of letters that are alphabetically smaller than a in T, then

$C(a) = v + 1$. Let $Occ(a, i)$ denote the number of times the character a has been seen in $B_T[1, i]$. To facilitate operations, the sentinel symbol, \$, is also included in C and Occ, where \$ is alphabetically smaller than all letters. Using $C(a)$ and $Occ(a, 1)$, Ferragina and Manzini proposed an algorithm for searching a query Q in T. Let S denote a string with the suffix array interval in the range $[R_l, R_u]$. Now, for aS the new suffix array interval can be computed from $[R_l, R_u]$ using Occ and C as follows:

$$
\begin{aligned}
R_l &= C(a) + Occ(a, R_l - 1) \\
R_u &= C(a) + Occ(a, R_u) - 1
\end{aligned}
\tag{1.2}
$$

and $R_l(aS) \leq R_u(aS)$ if and only if aS is a substring of T. This means $O(|Q|)$ time is required to check if Q is a substring of T and to count the number of times Q has been seen in T. This is called backward search.

1.2.4 Contiguous seed

Contiguous seed is a run of matches between sequence reads. Contiguous seed similarity search is based on the fact that a long substring shared between query and target sequences may be originated from significant similarity [76]. The consecutive match of few bases is defined as the *seed* or *hit*. After identifying seed(s) between query and target sequences, the search is continued by extending the seeds regions up to a threshold measure of similarity to report the candidate positions. The most popular tool for similarity search, BLAST (Basic Local Alignment Search Tool) [3], is based on the contiguous seed similarity search for finding local alignment. The

length of the shared substring between sequences, seed, is w=11 by default. The target and query sequences should have w consecutive identical characters, or w matches, to be aligned, and is shown as seed in the form of 11111111111, where a 1 represents a match. The key idea here comes from the fact that searching exact matches would be easier than looking for approximate matches.

1.2.5 Spaced seed

Spaced seed similarity search, proposed by Ma, et.al. in 2002 [65] looks for matches that are not consecutive and can include some "don't care" positions, resulting in a higher probability of catching alignments. The spaced seed proposed in [65] is $111 * 1 * *1 * 1 * *11 * 111$, where a 1 denotes for a match and $*$ corresponds to a "don't care" location. It means, when looking for a possible alignment, only locations of 1's are examined, whereas locations with *'s are not checked. As opposed to the consecutive seed of BLAST, this class of seeds is referred as *spaced seed*. Spaced seeds are the current state-of-the-art for approximate sequence matching, which have been increasingly used to improve the quality and sensitivity of searching in different applications including alignment-free methods and metagenomic studies [29, 30, 39, 84]. Spaced seeds are more sensitive and have higher chance to identify actual alignments. The *weight*, denoted by w, of a spaced seed is defined as the number of 1's in the seed whereas the total number of 1's and *'s is defined as the *length* and denoted by l. A seed's capability to identify regions that are similar between sequences is called the *sensitivity* of that seed [76]. There is a dynamic programming algorithm for calculating

the sensitivity of a spaced seed [57]. Using a set of several spaced seeds for similarity search will greatly improve the sensitivity. This set of several spaced seeds is referred as *multiple spaced seeds* [76].

There are some limitations when using spaced seeds for approximate string matching. The first one is designing and finding the optimal seed or multiple seeds. This is an NP-hard problem (there are exponential number of candidate sets of spaced seeds to examine, and computing the sensitivity of each spaced seed set also needs exponential time), and it has been addressed in the several previous works [23]. The second issue related to spaced seeds is the large space requirements for spaced seeds index table that makes it infeasible to apply them for large-scale alignment tasks. Recently, Egidi and Manzini [23] have proposed a class of spaced seed called, Quadratic Residue seeds (QR-seed), having competitive sensitivity/specificity to the best multiple spaced seeds. It is based on one spaced seed, and requires less memory compared to multiple spaced seeds.

1.3 Sequence alignment

Sequence alignment is one of the essential operations in computational biology and bioinformatics, and is affected adversely by the NGS data deluge. In genome resequencing projects, it is often used prior to variant calling. In transcriptome resequencing, it provides information on gene expression measurements. It is even used in *de novo* sequencing projects to help contiguate assembled sequences. As such, improving the performance of alignment algorithms would have a wide effect in the field.

Formally, sequence alignment problem is defined as follows. As input we are given a set of target sequences T, and a set of query sequences S, and the goal is to find one or more coordinates from each query sequence in the set of target sequences based on the similarity between sequences. Given a distance function F, and an error threshold e, we are looking for the set of positions in T where a query sequence has a distance $\leq e$ under the distance function F. Common distance functions are *hamming* distance, corresponding to the number of mismatches, and *edit* distance, corresponding to the number of mismatches, insertions, and deletions. The best output record of a given query $s \in S$ is the position in the target sequence $t \in T$ that is the most probable position that s originated form. The measures that can be used to define this likelihood is the minimum number of edit operations or the minimum sum of *Phred* quality score of error positions in s. The Phred quality score Q is defined as a measure of quality related to the base-calling error probabilities P, and is equal to $Q = -10\log_{10}P$ [24]. For instance, if Phred score of a given base is 20, the probability that the given base is identified incorrectly is equal to 0.01.

Sequence alignment is broadly categorized into two classes: local alignment and global alignment. A local alignment, instead of considering the total sequence, takes into account regions of all possible lengths, and detects similar regions between two sequences. Whereas in global alignment all the bases in two sequences are involved and taken into account in the alignment process. The most widely used algorithms for solving the global and local alignment problems are based on dynamic programming approaches [83, 100]. Creating the dynamic programming alignment matrix and perform-

ing a backtrack in it to detect the optimal alignment requires $O(nm)$ time. The dynamic programming approaches are slow due to their quadratic time complexity, and will take long time especially in cases where we have vey long target sequences to be compared such as large genomes [76].

Finding exact occurrences of queries in target sequences can take linear time, however a linear time complexity in order of the length of the long sequence or target is too expensive, so the time complexity linear in the size of the short sequence or query is desired. To achieve this goal, an index such as a suffix array, suffix tree, FM-index, or hash table can be constructed on the large sequence . Building such an index requires $O(n)$ time and $O(n\log n)$ memory, where n is the length of the target sequence. The indexing cost is usually amortized when the index is utilized multiple times, resulting in quick lookup of the indexed sequences. During the past decade, many alignment methods have been proposed based on suffix trees [47, 73], suffix arrays [1, 37], hash tables [3, 12, 34, 38, 56, 63, 64, 93, 99, 108], or FM-indices [48–50, 52, 54, 55, 59, 69]. Below is a summarized description of some of the most popular alignment methods in the field.

1.3.1 BWA, BWA-SW, BWA-MEM

BWA [55] is an FM-index based alignment method. For exact string matching, it uses the FM-index and backward search. For inexact matching, it utilizes a heuristic bounded traversal/backtracking approach. Using the proposed recursive approach, BWA identifies the intervals related to the suffix array of the target's substrings matching the query sequence q with less than e gaps or mismatches. This procedure is performed by backward search to

sample distinct subsequences from the target sequence which is bounded by the array $D(.)$, where $D(i)$ is the lower bound on the count of different bases between prefix $q[0;i]$ and the target sequence. Calculating better estimates for D narrows down the search space and results in a faster and more efficient process.

BWA-SW [54] constructs FM-index for the target and query sequence. The method represents the target sequence as a prefix trie implicitly and describes the query string in a prefix directed acyclic word graph (DAWG), reconstructed from the prefix trie of the query string. The proposed method then applies a dynamic programming approach between the DAWG and the trie, by traversing the query DAWG and the target prefix trie. Instead of outputting all the significant local alignments, the algorithm only outputs largely non-overlapping mappings on the query sequence. BWA-SW heuristically finds and ignores seeds contained in a longer alignment, which results in less computing time for extension of unsuccessful seeds [54].

BWA-MEM [52] is a seed-and-extend based alignment method. The proposed method works by first seeding an alignment using super-maximal exact matches, called SMEMs, by utilizing an algorithm that looks for the longest exact match spanning positions in a given query. Nevertheless, the actual alignment may not sometimes have any SMEMs. To decrease misalignment resulted from absent seeds, BWA-MEM uses re-seeding. Suppose we have a SMEM of length l that happens k times in the target sequence. If the length of SMEM, l, is too long, longer than 28bp by default, BWA-MEM re-seeds with the largest exact matches that span the middle nucleotide of the SMEM and appear at least $k+1$ times in the target [52]. These kinds of

seeds can be identified by demanding an occurrence threshold in the original SMEM method.

1.3.2 Bowtie, Bowtie2

Bowtie [50] is an FM-index based alignment algorithm. After building the index for target sequences, it employs a modified Ferragina and Manzini [26] exact matching approach to determine the possible mapping coordinates. To handle inexact matching, Bowtie customizes this method by adding two extensions. The first one is a quality-aware backtracking procedure allowing mismatches, and the second one is "double indexing", to prevent excessive backtracking. Bowtie allows a limited number of mismatches and chooses mappings that have lower values for sum of the quality at all mismatched locations [50]. It launches a backtracking search to identify mappings that assure the above mapping procedure. The search continues similarly for the exact backward search, computing ranges for larger query suffixes. The algorithm may choose a current matched query location if it encounters an empty range, and changes the related nucleotide with another base. It then restarts the backward search from the next position after the changed nucleotide. Bowtie algorithm considers only those changes that are compatible and agree with the mapping policy resulting in a modified suffix that appears at least once in the target. To reduce excessive backtracking, this method utilizes "double indexing". For target sequences, two indices are constructed. The first one includes the BWT of the forward-strand target set, called forward index, and the second one consists of the BWT of the reversed-strand, not the reverse-complement, which is referred as the *mirror*

index.

Bowtie2 [49], as opposed to Bowtie, handles the gapped alignment by extending the FM-index based appraoch of Bowtie to allow gapped alignments by dividing the algorithm into two steps: (*i*) ungapped seed-finding step based on FM-index that utilizes the speed and memory efficiency of it and (*ii*) gapped extension step based on the dynamic programming approach that takes advantage of single-instruction multiple-data (SIMD) vector processing extensions of modern processors. To do so, the proposed method identifies seeds from the query sequence and its reverse complement. After seed extraction, Bowtie2 utilizes the FM-index to identify all the ranges related to these seeds in an ungapped fashion and then prioritizes them and calculates their positions in the reference genome from the index. Finally, it uses a SIMD-accelerated version of Smith-Waterman algorithm [25] to extend the seeds into full alignments.

1.3.3 Novoalign

Novoalign (`http://www.novocraft.com`) is a hash-based alignment algorithm. It provides the most sensitive results for the alignment problem by employing a dynamic programming approach. In the indexing stage, Novoalign constructs a hash table by partitioning the target sequences into overlapping k-mers. In the mapping phase, after constructing the index table, it exploits the Needleman-Wunsch dynamic programming approach [83] with affine gap penalties to identify the optimal global alignment.

1.3.4 GEM

GEM [69] is an FM-index based alignment method. It utilizes a filtration-based paradigm to approximate sequence matching. To do so, all related candidate matches are identified using an FM-index by appropriate pigeonhole-like rules followed by a dynamic programming approach for refining alignments. Some of the regions may result into many matches as candidates and require to be verified, and hence resulting in inefficient alignments. GEM picks a threshold t, which defines an upper bound for candidates that should be examined for each filtering region. For a given query, GEM scans it backward from right to left with FM-index, appending one base at a time to the current segment, and calculating the candidates count in the target that map exactly to the sequence being formed. Any moment the count of candidates goes under the threshold, it starts a new segment. It should be mentioned that the number of candidates to be examined per filtering region is always guaranteed to be less than the threshold.

1.3.5 SOAP, SOAP2, SOAP3

SOAP [59] is a hash-based alignment method. For aligning a query against the target sequences, SOAP allows a fixed number of mismatches (at most 2) or a single continuous gap (1-3 bp) with no mismatches in the flanking regions. Its criteria for best alignment results are either the smallest number of mismatches or a shorter gap. SOAP loads the target sequences into memory and constructs the seed index tables for the target sequences. For each query, it then builds seeds and searches for candidate hits in the

corresponding index table. Finally, it performs the alignment and reports the results. SOAP uses a look-up table to accelerate the alignment. To reduce the memory requirements, both the query and the target sequences are converted into 2-bit encoding for each base. A bit-wise EXCLUSIVE OR (XOR) comparison is performed on query and target sequences to find the number of different bases. Since gapped hits do not include mismatches, SOAP uses an enumeration approach that attempts to add a consecutive gap or delete a region at each possible site in a query.

SOAP2 [60] utilizes Burrows Wheeler Transform (BWT) compact index to construct index from the target sequences in the main memory. By employing BWT, SOAP2 improves alignment speed and significantly reduces memory requirements. SOAP2 constructs a hash table to speed up searching for the location of a query in BWT target index. Because of the hash, detecting the actual location inside the block requires very few look-up interactions. For alignments with both indels and mismatches, it partitions the query into segments. To allow a single mismatch, a query is partitioned into two regions. For handling two mismatches, it partitions a query into three regions to look for hits. This enumeration approach was utilized to find mutation positions on the queries.

SOAP3 [62] is a parallel implementation of SOAP2 adapted for the Graphics Processing Unit (GPU) to improve the speed. It constructs a GPU variant of the compressed BWT based index used by SOAP2. To adapt the compressed BWT index with GPU, SOAP3 redesigns the BWT to decrease memory accesses to the full, while preserving the index efficiency. It also restricts the patterns introducing too many branches at runtime and

postpones the execution of them to decrease the idle time of processors.

1.4 Sequence assembly

In bioinformatics, sequence assembly process refers to joining small sequences, called reads, of a much larger and unknown DNA sequence in order to rebuild the initially long sequence. It is a required step in bioinformatics because the current sequencing technologies cannot generate whole genomes (a huge sequence, *e.g.* about 3 billion characters long for human) from start to end, but rather generate small sequences (reads) between 100 and 5000 bases long (Fig. 1.1: a,b). Then the assembly algorithm stitches together the genome from short sequenced pieces of DNA (Fig. 1.1: c-e). Therefore, given a collection of reads, *i.e.* short subsequences of the genomic sequence in the alphabet {A, C, G, T}, the goal of genome assembly is to completely reconstruct the genome from which the reads are derived. There are several challenges for the assembly problem such as sequencing errors, repeats and duplications. Also, large genomes require more computational power as well as memory (most algorithms require more than 300 GB memory for mammalian genomes) to assemble billions of reads. Assembly algorithms can be divided into three major paradigms: Greedy, Overlap-Layout-Consensus (OLC), and de Bruijn graph [81].

In greedy paradigm, the assembly algorithm always makes the best immediate local choice, *e.g.* the reads that have the best overlap are joined incrementally while they are in agreement with the recent built assembly. Here, the global relationship between the reads is not taken into account and

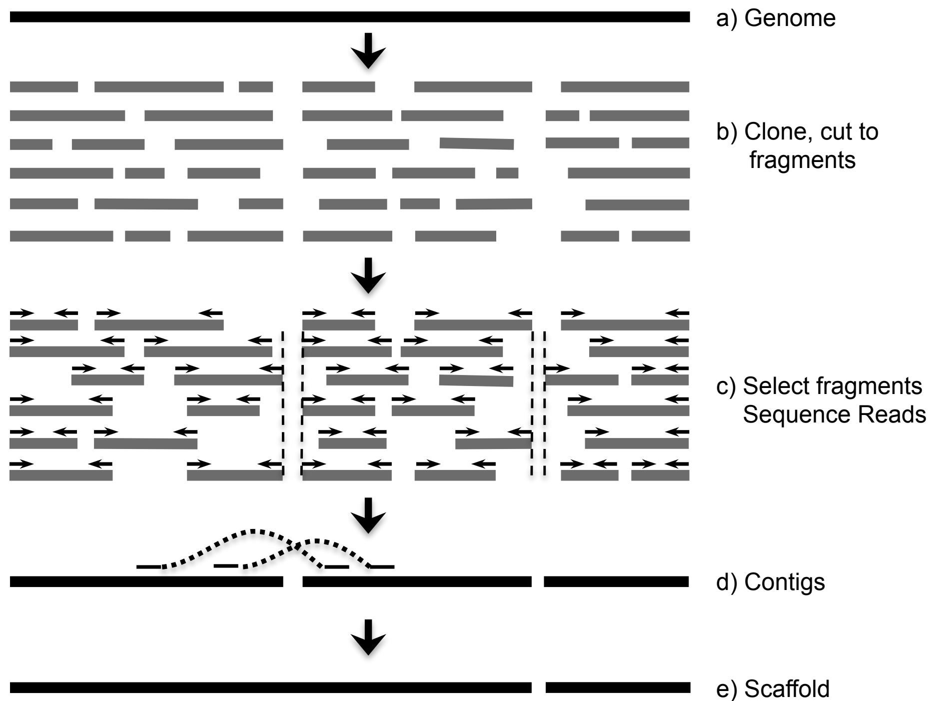

Figure 1.1: Genome sequencing and assembly steps.

all choices taken by the assembly method are inherently local. This paradigm has also challenges with assembling of large and complicated genomes.

In the Overlap-Layout-Consensus paradigm, the assembler first finds potentially overlapping reads (*overlap*), then merges reads into contigs and contigs into scaffolds (*layout*), and finally derives the DNA sequence and correct read errors (*consensus*). Because the complexity of overlap step is $O(n^2)$, this paradigm is not efficient for huge assembly problems such as human genome assembly.

In the de Bruijn graph paradigm, sequences are decomposed into fixed-length substring of length k, called k-mers. Then, each k-mer is assigned

to a vertex in the graph and the edges between vertices illustrate that the adjacent k-mers overlap by $k-1$ characters. The assembly algorithm then tries to join vertices. Hence, assembly task can then be observed as joining vertices of the graph when they are connected unambiguously. Before concatenating vertices, the graph must be cleaned of edges and vertices resulting from sequencing errors [98]. After creating all the k-mers from the input reads, the graph is constructed and finally the potential long sequences (genomes) are obtained by finding the Eulerian tours in the de Bruijn graph. Several methods exist for the sequence assembly problem.

Popular assemblers based on the OLC paradigm are Edena [36] and SGA [97] and for de Bruijn graph paradigm we can mention ALLPATHS [11, 31, 66], ABySS [98], Euler [86], SOAPdenovo [58], and Velvet [109]. Following is a summarized description of some of the assembly methods.

1.4.1 ALLPATHS

ALLPATHS [11] is a unipath graph-based assembly method. A unipath is an unambiguous path in the graph and a unipath graph is a graph whose edges are unipaths. Unipaths are found by traversing the nodes until a branching is found. One end of a unipath is arbitrarily named as the left side and the other end is named as the right side. For each unipath in the graph, ALLPATHS finds its neighbors on the left side and also on the right side. If the distance between the left neighbors and the right neighbors is less than a threshold then the unipath is removed. The remaining unipaths in the graph are seed unipaths. ALLPATHS starts assembly from low copy count seed unipaths called ideal seeds. For each ideal seed unipath, ALLPATHS

defines its neighborhood based on the distance on each side and constructs two sets of read clouds: the primary read cloud consists of the reads whose actual genomic locations are most likely near the seed unipath (reads on other ideal seed unipaths in the neighbourhood) and the secondary read cloud consists of all other short-fragments in the neighbourhood [11]. The reads in the neighborhood of an ideal seed unipath define a local unipath. Local unipaths are joined iteratively using mate pairs in the primary cloud, which in the end yield a sequence graph representing the genome. Then, ALLPATHS simplifies dead-ends, bubbles and loops to further simplify the graph.

1.4.2 SGA

The String Graph Assembler (SGA) constructs an assembly string graph to represent the reads. It first removes duplicate and contained reads from the input set of reads to build an overlap graph. Then transitive edges are removed resulting in an overlap graph of irreducible edges, called a string graph [97]. SGA uses an FM-index to directly compute the set of irreducible edges for a given set of reads. Reads in SGA are corrected based on the k-mer frequencies and approximate overlap between reads. SGA tries to correct the reads by minimizing the sum of edit distances of all overlapping reads in the string graph. After filtering and correction, most of the remaining reads do not contain errors. Each node in the string graph denotes a read. Most of the reads in the string graph will have only two neighbours, one overlapping with a prefix and the other overlapping with a suffix of the read. The majority of the read sequences in the string graph

are simply connected on a path without any branching. These reads can be unambiguously joined together to dercrease the size of the graph [97]. SGA iteratively discovers irreducible edges in the graph and reduces the nodes that are simply connected. Each edge in the simplified string graph represents a contig. In the last stage of assembly, paired-end or mate-pair information are employed to bridge contigs to form scaffolds by a program called scaffolder. The scaffolder constructs a contig linkage graph that represents relationships between contigs using mate pairs. The scaffolder then removes ambiguous edges from the contig linkage graph, where the contigs cannot be ordered consistently. Finally, terminal vertices (*i.e.*, vertices having an edge in only one direction) are identified and all paths between pairs of vertices are determined. The path with the largest sequence amount is retained as a scaffold [97].

1.4.3 ABySS

ABySS (Assembly By Short Sequences) is a parallel sequence assembly algorithm implemented using MPI and developed to assemble huge amounts of NGS data [98]. The ABySS algorithm has two stages. In the first stage, all k-mers are created from the reads and refined to remove read errors and build initial contigs. In the second stage, contigs are extended by resolving ambiguities using mate-pair information. Some genomes have a very large number of k-mers. For instance, human genome has over 2 billion unique k-mers. If every k-mer is represented using, say 50 bytes, 100 GB RAM is required just to represent k-mers. So, the solution of ABySS is distributed computing which distributes the de Bruijn graph over a cluster of compute

nodes as follows. Each input read of length l is divided into $l-k+1$ successive k-mers by shifting a fixed-length window with length k over the input read sequence [98]. Then, the cluster compute node index of each k-mer is computed, based on binary representation value of its sequences, and the k-mer is given to that node and stored in a hash table. After the k-mers were loaded into the de Bruijn graph which is distributed across several nodes, the adjacency information for the k-mers is calculated. To do so, a message is sent to 8 possible neighbors for each k-mer because a node in graph may connect up to 8 edges, one for each possible single-base extension of {A, C, G, T}, in both directions. If the neighbor is present, a $k-1$ overlap with the originating k-mer is available, and the adjacency information is updated respectively [98]. The de Bruijn graph may have false branches caused by noises and sequencing errors or bubbles caused by repeat read errors. In the next step, the graph is cleaned of those vertices and edges related to noises and sequencing errors (Trimming) as well as repeat read errors or haplotypic variants (Bubble popping). After removing ambiguous edges, the remaining de Bruijn graph is analyzed for contig extension and then vertices linked by unambiguous edges are merged together resulting in the initial contigs. After creating initial contigs, paired-end information is employed for resolving the ambiguities among contigs. Contigs that can be linked together are identified by paired-end reads. Then, the linked contigs are filtered to eliminate erroneous links resulted from misaligned or mispaired reads. Finally, contigs are considered to be linked if at least a predefined number of pairs join them together.

New releases of ABySS address challenges with the increase in read

length in the third generation of sequencing technologies. In this method, a new spaced seed data structure and algorithm has bee designed for paired de Bruijn graph genome assembly [6, 7]. The paired de Bruijn graph method is easily adaptable into any de Bruijn graph-based framework. This modified approach deems to be more fit than its predecessor due to the following reasons (i) Allows for sequence error corrections not only in low coverage regions but also across the entire genome due to the usage of two k-mers instead of one; (ii) Longer and unambiguous contigs can be reported from repeat-rich sequences; (iii) More unique k-mers pairs can be reported for larger separation distances using small k-mers (k'), which is only possible to reproduce if a single k-mer length is more than $2k'$. There are lesser nodes in the same de Bruijn graph due to the accommodation of two k-mers in each of them and less tangled edges due to the unique nodes. As an apparent consequence, the amount of memory and time efficiency is a lot lesser compared to a typical de Bruijn graph structure.

1.5 Overview of research goals

In Chapter 2, I introduce my technical notation and definitions then describe a distributed and parallel framework for large-scale sequence alignment tasks. This will address the memory requirement challenge for performing large-scale alignment tasks. The method that I develop is similar to MapReduce framework developed by Google for large scale data processing. It also employs the Bloom filter data structure at its core. As this is a very compact data structure, its peak memory footprint is much smaller

than other data structures. This will allow the reduction of the memory bottleneck of indexing large target sequences.

In chapter 3, I describe a fast hashing algorithm for bioinformatics applications called ntHash. This algorithm is a recursive or rolling hashing function in which the new hash value is derived from the previous one by few more operations. I demonstrate the uniformity and runtime performance of the proposed method on real sequencing data as well as simulated random data. Also, I compare the performance of ntHash with the state-of-the-art hashing algorithms in bioinformatics.

In chapter 4, I propose a streaming algorithm for cardinality estimation in massive genomics data called ntCard. I will address this problem by utilizing the ntHash algorithm to efficiently compute hash values for input stream sequences. I will then sample the calculated hash values to build a reduced representation multiplicity table describing the sample distribution, and then derive a statistical model to reconstruct the population distribution from the sample distribution. I evaluate the performance of ntCard on whole genome shotgun sequencing experiments on the human genome and the white spruce genome datasets. I also compare the accuracy, runtime, and memory usage of ntCard to leading cardinality estimation algorithms. Finally, in chapter 5, I present concluding remarks.

Chapter 2

DIDA: Distributed Indexing Dispatched Alignment

2.1 Publication note

The work described in this chapter was previously published in the *PLoS One* journal in 2015 [79]. The outcome algorithm and software tool from this work was used in two published papers in *the Plant Journal* in 2015 [106] and the *Genome biology and evolution* journal in 2016 [43]. The work described in this chapter is the main work of the author, under the supervision of his PhD supervisor, Inanc Birol. The author designed the algorithm, implemented the software tool, performed the experiments, and wrote the manuscript. The co-authors on this work helped in optimizing the software tool and performing the experiments.

2.2 Author summary

Sequence alignment is one of the most popular and widely-used applications in bioinformatics that is affected by the high-throughput sequencing data deluge. In sequence alignment process, nucleotide or amino acid se-

quences are queried against targets to find regions of close similarity. The alignment process becomes computationally challenging when queries are too many and/or targets are too large. This bottleneck is usually addressed by preprocessing techniques, where the queries and/or targets are indexed for easy access while searching for matches. When the target is static, such as in an established reference genome, the cost of indexing is amortized by reusing the generated index. However, when the targets are non-static, such as contigs in the intermediate steps of a *de novo* assembly process, a new index must be computed for each run. To address such scalability problems, we present DIDA, a novel framework that distributes the indexing and alignment tasks into smaller subtasks over a cluster of compute nodes. It provides a workflow beyond the common practice of embarrassingly parallel implementations. DIDA is a cost-effective, scalable and modular framework for the sequence alignment problem in terms of memory usage and runtime. It can be employed in large-scale alignments to draft genomes and intermediate stages of *de novo* assembly runs. The DIDA source code, sample files and user manual are available through `http://www.bcgsc.ca/platform/bioinfo/software/dida`. The software is released under the British Columbia Cancer Agency License (BCCA), and is free for academic use.

2.3 Introduction

Performing fast and accurate alignments of reads generated by modern sequencing technologies represents an active field of research. At its core, the

sequence alignment problem is about identifying regions of close similarity between a query and a target. Most modern algorithms in this domain work by first constructing an index of the target and/or the query sequences. This index may be in the form of a suffix tree [47, 73], suffix array [1, 37], hash table [3, 12, 34, 38, 56, 65, 93, 99, 108], or full-text minute-space index (FM-index) [48–50, 55, 58, 61, 69]. Although this pre-processing step introduces an initial computational overhead, indexing helps narrow the list of possible alignment coordinates, speeding up the alignment task.

When the target sequence is static (e.g., a reference genome), the cost of index construction represents a one-time fixed-cost. It is performed once as a pre-alignment operation, and the resulting index is used for many subsequent queries. Hence, it is often discounted in performance measurements of alignment tools. However, there are many applications where the reference is not static and/or the computational cost of indexing is not negligible. Such cases include resequencing data analysis of non-model species, where the target index has to be established, and intermediate stages of a *de novo* assembly process where index construction needs to be performed several times.

One more complicating factor in these two domains is that the target sequence may not represent chromosome-level contiguity, requiring alignments to a fragmented target sequence. This may be a particular challenge for many alignment algorithms, which perform poorly near target boundaries, introducing “edge effects”.

To address these challenges, we have designed and developed DIDA, for Distributed Indexing and Dispatched Alignment. DIDA works by first dis-

tributing the index construction over several computing nodes. It dispatches the query sequences over corresponding computing nodes for alignment. Finally, partial alignment results from different computing nodes are gathered and combined for reporting.

We tested DIDA using four datasets: (1) *C. elegans* genome, (2) Human draft genome, (3) Human reference genome, and (4) *P. glauca* genome. Here, we report on the scalability, modularity and performance of the tool.

2.4 Method

The sequence alignment task is often suitable for parallel processing, and the widely practiced approach is to perform the task in an embarrassingly parallel manner. When the target index is available, it is loaded on multiple processors, and a subset of the query sequences (usually raw reads from a sequencing experiment) are aligned in parallel to this common target.

In DIDA, we parallelize both the indexing and alignment operations using a five-step workflow (Fig. 2.1). For the description of the method, we consider a use case where the target is a draft genome assembly, with individual contigs and scaffolds related to each other through an assembly graph. Although, the protocol is general enough for a generic target not associated with a graph. Before describing the proposed framework, we provide some preliminary and basic definitions and notation.

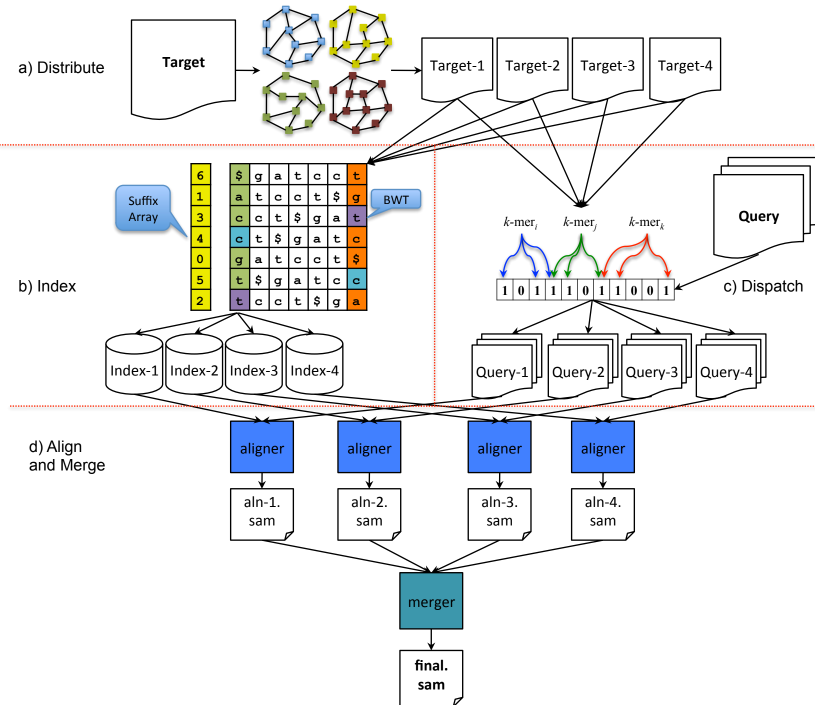

Figure 2.1: The workflow of DIDA with four partitions as an example. (a) First, we partition the targets into four parts using a heuristic balanced cut. (b) Next, we create an index for each partition. (c) The reads are then flowed through Bloom filters to dispatch the alignment task to the corresponding node(s). (d) Finally, the reads are aligned on all four partitions and the results are combined together to create the final output.

2.4.1 Assembly graph

Most modern assembly tools are graph-based algorithms [11, 58, 80, 97, 98, 109]. These algorithms model the assembly problem using a graph data structure (e.g., *de Bruijn* graph, overlap graph, string graph) consisting of a set of vertices (reads or k-mers) and edges (overlaps) representing the relationship between vertices. After building such graphs, the assembly problem is converted to a graph traversal problem, where a walk along the

graph would reconstruct the source genome. In practice, assembly algorithms report unambiguous walks on the assembly graphs, building contigs as opposed to full genomes or chromosomes. Especially for large genomes, use of short reads from high-throughput sequencing platforms for assembly results in a large number of contigs. For example, using 150 base pairs (bp) reads, the white spruce genome is assembled into several million contigs [8].

Some of the ambiguity on the assembly graph can be mitigated by using paired end reads or other linkage information. This requires alignment of queries to a typically highly fragmented draft genome. In DIDA, when available, we partition the assembly graph keeping tightly connected components on the same partition, as described below. For other use cases, where the target components (e.g., contigs or chromosomes) are not related through a graph, the partition optimization is done based on component lengths.

2.4.2 Bloom filter

As explained in the previous chapter, a Bloom filter [9] is a space-efficient and compact probabilistic data structure providing membership queries over dynamic sets with an allowable false positive rate. Bloom filter has been widely used in many computing applications, which exploit its ability to succinctly represent a set, and efficiently filter out items that do not belong to the set, with an adjustable error probability. In bioinformatics, the Bloom filter has been recently utilized in applications such as k-mer counting, genome assembly and contamination detection [14, 16, 74, 90, 102].

An ordinary Bloom filter consists of a bit array B of m bits, which are initially set to 0, and φ hash functions, $h_1, h_2, \ldots, h_\varphi$, mapping keys from

a universe U to the bit array range $\{1, 2, \ldots, m\}$. In order to insert an element x from a set $S = \{x_1, x_2, \ldots, x_n\}$ to the filter, the bits at positions $h_1(x), h_2(x), \ldots, h_\varphi(x)$ are set to 1. To query if an element q is in the filter, all of the bits at positions $h_1(q), h_2(q), \ldots, h_\varphi(q)$ are examined. If at least one bit is equal to 0, then q is definitely not in S. Otherwise, q likely belongs to the set. The uncertainty stems from coincidental cases, where all the corresponding bits, $h_i(q)$ $i = 1, 2, \ldots, \varphi$, may be equal to one even though q is not in S. This is possible if other keys from S were mapped into these positions. Such a chance occurrence is called a false positive hit, and its probability is called the false positive rate, F. The probability for a false positive hit depends on the selection of the parameters m and φ, the size of the bit array and the number of hash functions, respectively. After inserting n distinct elements at random to the bit array of size m, the probability that a specific bit in the filter is 0 is $(1 - \frac{1}{m})^{\varphi n}$. Therefore, the false positive rate is:

$$F = (1 - (1 - \frac{1}{m})^{\varphi n})^{\varphi} \approx (1 - e^{-\varphi/r})^{\varphi} \tag{2.1}$$

where $r = m/n$ is the number of bits per element. Minimizing the equation (1) for a fixed ratio of r yields the optimal number of hash functions of $\varphi = r\ln(2)$, in which case the false positive rate is $(0.6185)^r$ [10].

2.4.3 DIDA

Our proposed distributed and parallel indexing and alignment framework, DIDA, consists of five major steps to perform the indexing and alignment

task: distribute, index, dispatch, align, and merge. The indexing and dispatch steps are performed in parallel. Each step is explained in detail as follows.

Distribute. In this step, the set of target sequences is partitioned into several subsets. Depending on the nature of the target sequences (static as in reference genomes, or non-static as in a draft assembly; unrelated as in chromosomes, or related as contigs in an assembly graph) different partitioning strategies may apply to initial target set. The key point in all cases is to keep the partitions as balanced as possible. The target partitioning problem is a variant of the bin-packing problem and since the bin-packing problem is NP-hard, there is no polynomial time solution to the target partitioning problem either. However, there are efficient heuristics developed to solve the problem [44, 105]. Other than the theoretical hardness of the target partitioning problem, having well-balanced partitions in practice when the target set contains few number of long sequences will also be difficult.

In the case of a static and independent set of target sequences, the partitioning is performed using the best fit decreasing strategy that is among the simplest greedy approximation algorithms for solving the bin-packing problem. Here, the bins correspond to computing nodes, and items are the target sequences. This strategy operates by first sorting the target sequences to be partitioned in decreasing order by their lengths, and then distributing each target sequence into the best node in the list, which is the node with the minimum sufficient remaining space for the target sequence.

When the target sequences are related, such as contigs in a draft assem-

bly, the partitioning starts by first identifying all connected target sequences using adjacency information in the assembly graph. This is performed by launching a depth-first search traversal of the undirected adjacency graph, and by finding all connected components. Then, the partitioning procedure continues by applying the best fit decreasing strategy for identified connected components to distribute them over computing nodes.

Index. An exact pattern search can take linear time in the target length. But when the target length is very long, it is desirable to have the search time linear in the query length and independent of the target length. To do so, an index such as a suffix tree, suffix array [67, 68, 87], hash table, or FM-index [26, 27] on the long sequence or text can be created. Constructing such an index takes $O(n)$ time and $O(n\log n)$ space, where n is the size of the target [87]. This cost is often amortized when the index is used several times, providing very fast searching of the indexed target.

For the indexing step, and on each computing node, DIDA takes the subset of target sequences from the Distribute step, and constructs an index for each subset in parallel on all computing nodes. Then, the indices are stored on each computing node to be invoked later in the alignment step. Depending on the alignment algorithm, any indexing approach can be used in this step. The reduced target size (n/P in the best-case scenario, where P is the number of partitions) allows linear scaling of the indexing time and better than linear scaling in the index space. While both would have a positive impact on alignments against dynamic targets, the latter would also help cases where the target is too large to fit into the memory of a single

computer.

Dispatch. To keep track of each subset of target sequences, and to identify which read may align on which partition of the target, a set of Bloom filters is created for all partitions of target sequences. The reads are then *flowed* through these Bloom filters, and dispatched to the corresponding node(s).

To create a Bloom filter for each partition of target sequences, all target subsequences of length b (b-mer) in the partition are scanned. Each scanned b-mer, x, is then inserted into the corresponding Bloom filter by setting the related bits in the bit array, i.e. $B[h_i[x]] = 1, i = 1, 2, \dots, \varphi$. After constructing the Bloom filters, all possible read b-mers are queried against the Bloom filters. If at least one hit is found for each read, the read is dispatched to the corresponding node. This procedure continues until all the reads are either dispatched or discarded. By choosing the b-mer length, b, small enough, we make sure that no read sequence will be missed in the Bloom filter query step. This is performed by setting b less than or equal to the minimum seed or exact match length of aligners, l, for candidate hits. Detailed procedure for choosing b, loading and querying Bloom filters is explained below.

Loading Bloom filter. In this stage, all target sequences in the target set $T = \{T_1, T_2, \dots\}$ are scanned using a sliding substring of length b. For each target sequence T_i, all possible $|T_i| - b + 1$ b-mers are scanned and then inserted to the Bloom filter after specifying the corresponding bit vector positions by computing the φ hash values for each b-mer (Fig. 2.2).

Querying Bloom filter. In this step, all reads in the read set $R = \{R_1, R_2, \dots\}$

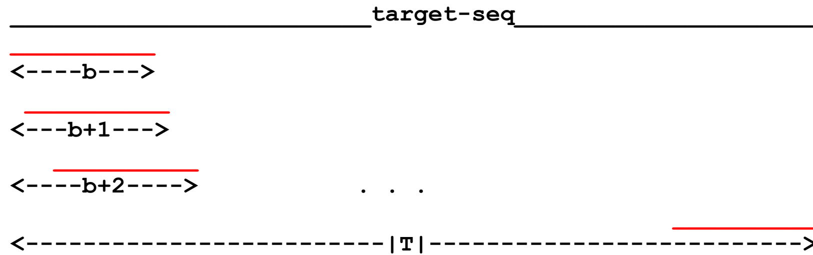

Figure 2.2: Loading the Bloom filter using a sliding substring of length b.

are queried using the same b-mer length in the loading stage. If at least one hit is found for a read, the read is dispatched to the corresponding node. We can use sliding b-mer windows (with step size of one base, $s = 1$) or jumping b-mer windows (with step size greater than one base) to interrogate each read R_j as explained in Fig. 2.3.

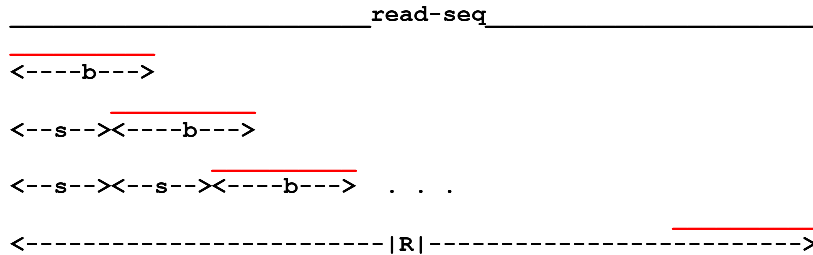

Figure 2.3: Querying the Bloom filter using a substring of length b.

Suppose that the minimum seed or exact match length to report a candidate hit for an aligner is l. We choose $b \leq l$ and then load the Bloom filters as mentioned in the *Loading Bloom filter* stage. In the querying stage, the b-mers of each read sequence is interrogated against all Bloom filters from different partitions. If $b = l$, the reads are scanned using sliding window with step size of one base, i.e. $s = 1$. By choosing $b < l$, the interrogation

step will be faster but with more extra dispatched reads, and the step size is computed as follows to cover all possible seeds or exact matches of length l between the target sequence T_i and read sequence R_j . If the l-mer starting at position p is covered by at least one b-mer, to cover the next l-mer starting at position $p+1$ we should have $s+b \leq l+1$. Therefore, $s \leq l-b+1$ (Fig. 2.4).

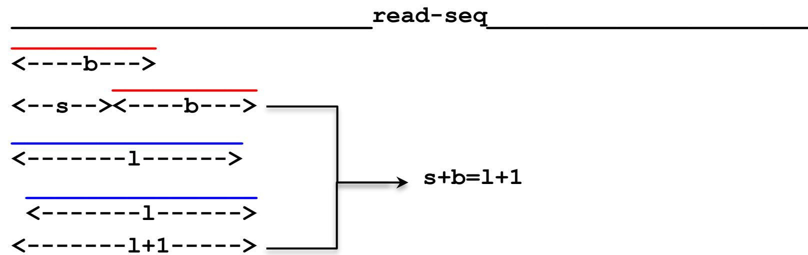

Figure 2.4: Choosing the jumping size s for querying the Bloom filter.

In the implementation, the values of r and φ can be set as input parameters and we have considered 8 bits for each b-mer, $r = 8$, as default. Therefore, the optimal number of hash functions that minimizes the false positive rate of Bloom filter is $\varphi = r\ln 2 \approx 5$, resulting in a false positive error rate slightly larger than 2%. It should be mentioned that the false positive rate does not affect the final alignment result. It only imposes more workload on nodes by dispatching reads that do not necessarily belong to those nodes as a result of false positive Bloom filter hits. Fig. 2.5 shows an example of how different values of r and φ affect the number of extra dispatched reads over multiple nodes.

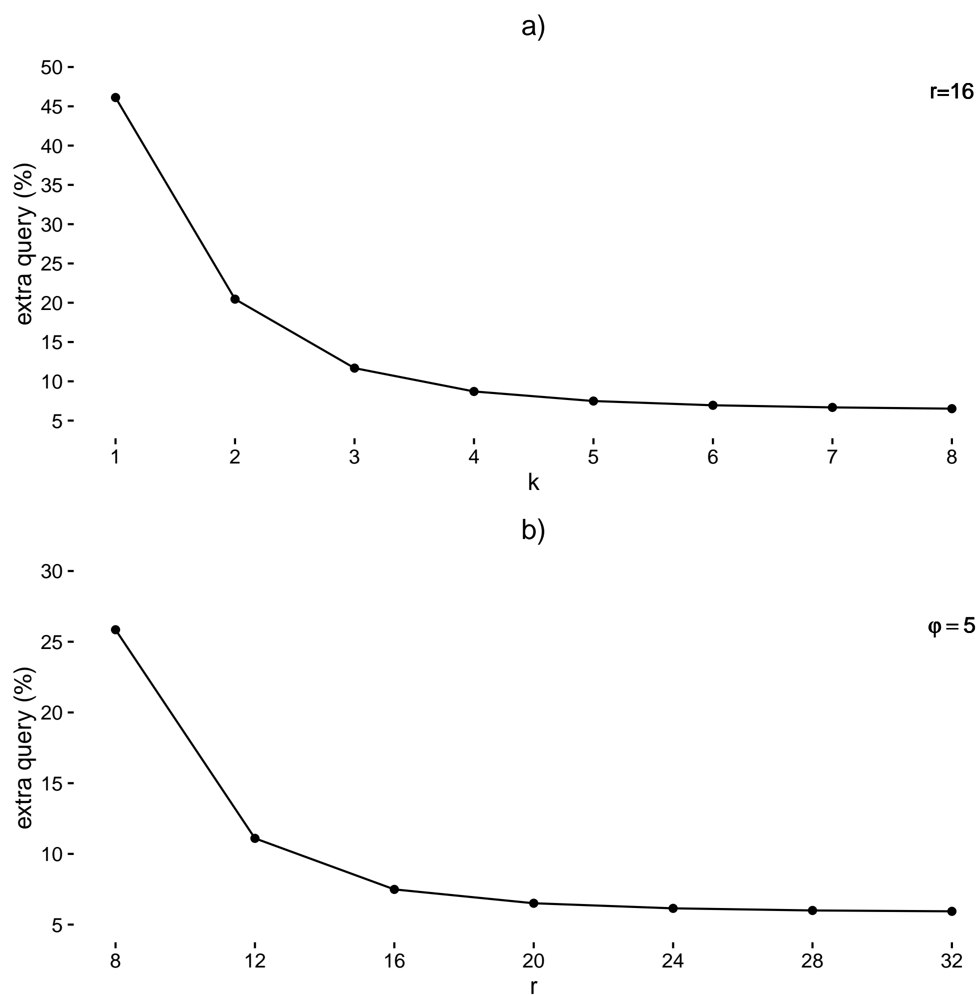

Figure 2.5: Extra dispatched reads vs. r and φ. (a) Percentage of the extra dispatched reads vs. the number of hash functions, φ, for the fixed value of $r = 16$ on the *C. elegans* dataset. (b) Percentage of the extra dispatched reads vs. the number of bits per item, r, for the fixed value of $\varphi = 5$ on the *C. elegans* dataset.

Align and Merge. After constructing indices for all sets of target sequences and dispatching the reads to the computing nodes, DIDA aligns the reads against the target sequence in parallel on each node. Note that, DIDA

itself does not offer an alignment algorithm; instead, it can use a variety of third party alignment tools.

In the merging step, partial alignment results, usually stored as SAM or BAM (Sequence Alignment/Map Format) [55] records from different computing nodes, are gathered and combined into the final SAM/BAM output. Depending on the aligner parameters for reporting the output, different merging approaches are applied. For example, when aligner parameters are adjusted in order to obtain the best unique mapped query, the merger will take into account that information to pick up the best quality mapped record for each query from the related records in all partitions. With the reporting parameters set to obtain multiple alignment records, the merger procedure searches for all or up to a predefined distinct number of alignment records in the partial alignment results in all partitions. The pseudo-code for DIDA is presented in Algorithm 2.1.

2.4.4 Implementation

DIDA is written in C++ and parallelized using OpenMP for multi-threaded computing on a single computing node. For distributed computing, DIDA employs Message Passing Interface (MPI) for inter-processor communications. As input, it gets the set of target sequences and the set of queries in FASTA or FASTQ formats, and the default output alignment format is SAM.

Program 2.1 The DIDA algorithm

```
1: Input: Set T of target sequences, set Q of query sequences
2: procedure DISTRIBUTE
3:     comp ← dfs(T)                              ▷ identifying connected components
4:     comp_sorted ← qsort(comp)                  ▷ sorting connected components
5:     P ← best-fit-decreasing(comp_sorted)  ▷ partitioning sorted connected components
6: procedure INDEX
7:     for all p ∈ P in parallel do
8:         index_p ← build-index(p)   ▷ constructing index for each partition
9:         store-index(index_p)             ▷ storing index for alignment step
10: procedure DISPATCH
11:     for all p ∈ P in parallel do          ▷ loading Bloom filter for each partition
12:         for all t ∈ p do
13:             for all b-mer ∈ t do
14:                 insert(b-mer,BF[p])
15:     for all q ∈ Q do                     ▷ flowing queries through partitions
16:         for all p ∈ P in parallel do
17:             if contain(b-mer ∈ q,BF[p]) then
18:                 dispatch(q,node_p)
19: procedure ALIGN
20:     for all p ∈ P in parallel do   ▷ aligning queries against targets on all partitions
21:         receive(q, node_p)
22:         s ← align(q, index_p)
23:         send(s, node_merger)
24: procedure MERGE
25:     while receive(s, node_i) in parallel do     ▷ merging results from all partitions
26:         insert(s, priority-queue)
27:         s ← priority-queue.pop()
28:         write(s, samFile)
29: Output: File samFile
```

2.4.5 Evaluated tools

To evaluate the performance of DIDA, four alignment tools have been used within the proposed framework: BWA-MEM [52], Bowtie2 [49], Novoalign (`http://www.novocraft.com`), and ABySS-map [98]. The first three algorithms were described in detail in Section 1.3.

Similar to BWA and Bowtie2, ABySS-map, a utility within the ABySS genome assembly software [98], constructs an FM-index for target sequences to perform exact matches. It is mainly used for alignment tasks in the intermediate stages of the ABySS assembly pipeline. In order to speed up the alignment operations, and hence the total assembly process, ABySS-map only performs exact matching and avoids backtracking for inexact matching.

All four tools are run with their default parameters, and the parameters related to the resource usage are set in a way to utilize the maximum capacity on each computing node. For example, all tools are run in multi-threaded mode with the maximum number of threads on each node. The performance of each alignment method is compared with itself within the DIDA framework.

Results were obtained on a high performance computer cluster consisting of 500 computing nodes, each of which has 48 GB of RAM and dual Intel Xeon X5650 2.66GHz CPUs with 12 cores. The operating system on each node is Linux CentOS 5.4. The cluster's network fabric and file system are Infiniband 40 Gbps and the IBM GPFS, respectively.

2.5 Results

2.5.1 Performance on real data

To evaluate the performance and scalability of DIDA on real data, we downloaded publicly available sequencing data on the *C. elegans* genome, a draft human genome assembly, human reference genome, and the *P. glauca* (white spruce) genome from the following websites.

- *C. elegans*:

  `http://www.ncbi.nlm.nih.gov/sra/?term=ERR294494`

- Human genome (NA12878):

  `http://www.nature.com/ng/journal/v43/n5/full/ng.806.html`

- Human genome reference (hg19, GRCh37):

  `http://hgdownload.cse.ucsc.edu/goldenPath/hg19/database/`

- *P. glauca* (accession number: ALWZ0100000000 and PID: PRJNA83435):

  `http:/www.ncbi.nlm.nih.gov/bioproject/83435`

In order to assess the performance of DIDA for each aligner on non-static targets, we assembled the reads from each dataset (except the human reference genome) using ABySS 1.3.7, and used the assembly graph in intermediate stages to guide partitioning. We have also evaluated the performance of DIDA for each aligner on human genome reference (hg19, GRCh37) as a static target. The detailed information of each dataset is presented in Table 2.1.

Table 2.1: Dataset specification.

Data	#targets	target(bp) length	#reads	read(bp) length
C. elegans	152,841	106,775,302	89,350,844	8,935,084,400
Human	6,020,169	3,099,949,065	1,221,224,906	123,343,715,506
hg19	93	3,137,161,264	1,221,224,906	123,343,715,506
P. glauca	70,868,549	35,816,518,982	1,079,576,520	161,936,478,000

Fig. 2.6 shows the scalability of wall-clock time and indexing peak memory usage of all four aligners on the *C. elegans* dataset, in standalone case (grey bars) or within the DIDA framework on two, four, eight, and 12 nodes, as indicated. For instance, for ABySS-map, the runtime of 2154 sec without DIDA is decreased to 893 sec using DIDA on four computing nodes. From Fig. 2.6 we can see the runtime scalability and modularity of different aligners within DIDA protocol. Notably, we have better scalability for the slower alignment tool, Novoalign. On memory usage, all aligners scale well within the DIDA framework. For example, the peak memory usage of ABySS-map indexing goes from 1100 MB without DIDA to 238 MB with DIDA on four computing nodes.

Table 2.2: Alignment time/indexing memory for all aligners on *C. elegans*.

	time/mem (sec/MB)			
node#	abyss-map	bwa	bowtie	novoalign
1	2154/1100	945/156	1700/274	19671/589
2	1261/522	667/80	1014/163	6305/263
4	893/238	574/65	737/99	5184/151
8	723/120	526/134	595/62	4788/69
12	700/81	547/89	601/46	4464/50

Tables 2.2-2.5 summarizes all results in the form of alignment-time/indexing-

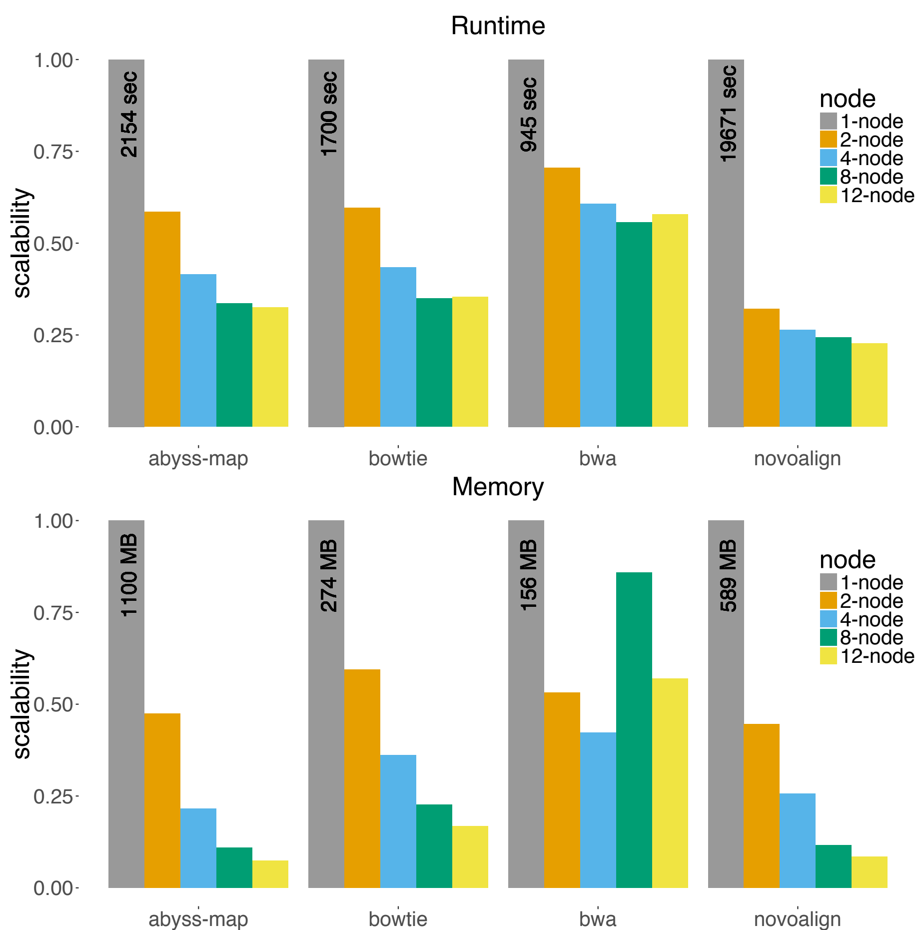

Figure 2.6: Scalability of different aligners using DIDA for *C. elegant* data. Y-axis indicates the runtime/memory scalability in the in the [0..1] interval for different alignment tools. The scalability of each tool is shown in the standalone case and within DIDA framework on 2, 4, 8, and 12 nodes [79].

memory. Regarding the accuracy of the alignment results for all aligners within DIDA framework on multiple nodes, we have compared them with the baseline results and found the accuracy of results the same as expected. As mentioned in the previous section, by choosing the b-mer length small enough, we make sure that no potential alignment is missed in the dispatch

Table 2.3: Alignment time/indexing memory for all aligners on human draft assembly.

	time/mem (min/MB)			
	abyss-map	bwa	bowtie	novoalign
1-node	652/31000	407/4400	1174/6100	59125/9300
2-node	472/15000	254/2200	611/2900	35728/4100
4-node	343/8100	216/1100	493/1600	23311/3700
8-node	253/4100	191/559	371/977	17485/2100
12-node	210/2700	181/372	296/590	13141/1200

Table 2.4: Alignment time/indexing memory for all aligners on hg19.

	time/mem (min/MB)			
	abyss-map	bwa	bowtie	novoalign
1-node	444/33823	379/4709	996/5528	NA
2-node	323/16911	254/2354	512/3042	NA
4-node	232/8455	205/1177	352/1417	NA
8-node	173/4227	171/588	254/667	NA
12-node	160/3170	164/441	226/495	NA

Table 2.5: Alignment time/indexing memory for all aligners on *Picea glauca*.

	time/mem (min/GB)			
	abyss-map	bwa	bowtie	novoalign
1-node	NA	NA	NA	NA
2-node	1201/184	NA	NA	NA
4-node	827/81	NA	NA	NA
8-node	638/45	NA	NA	NA
12-node	574/31	NA	NA	NA

step, and therefore, the accuracy of final alignment results will not be degraded. For example, the number of aligned reads for total 89,350,844 reads in the *C. elegans* dataset using ABySS-map within DIDA on 2, 4, 8, and 12 nodes is 86,851,694 which is almost the same as in baseline or standalone mode with the same SAM/BAM quality scores.

One point that should be addressed is that by increasing the compu-

tational power, *i.e.,* number of computing nodes, we may not necessarily obtain better runtime scalability due to the related overhead of the dispatch and merge steps. In general, for any parallel algorithm the runtime scalability should follow the behaviour shown in Fig. 2.7.

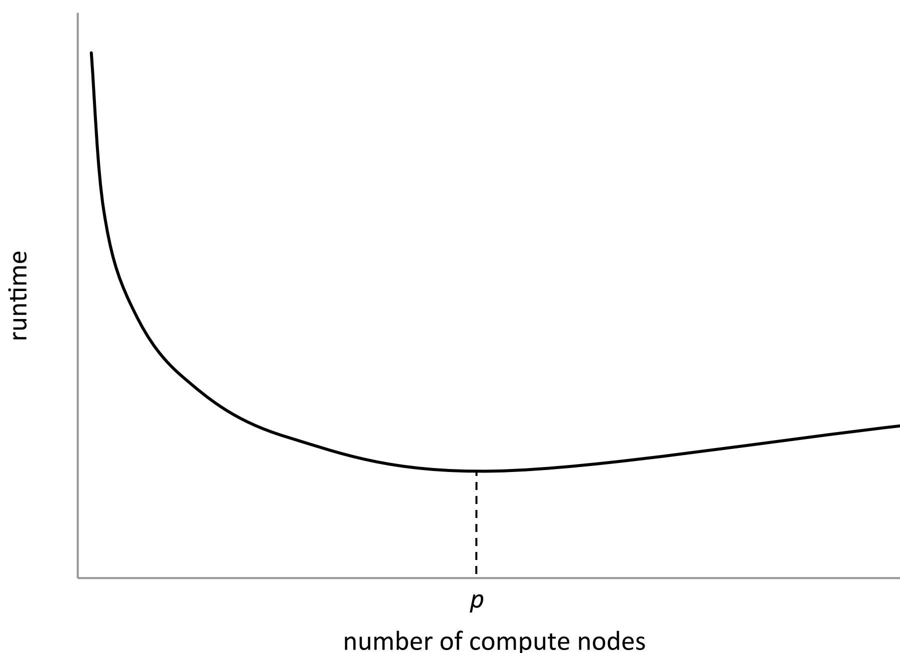

Figure 2.7: Runtime scalability behaviour with the number of nodes.

The optimal value for the number of nodes, p, may be different for each dataset and each algorithm. For example, for a small dataset such as *C. elegans*, with the BWA and Bowtie2 aligners, p is between 8 to 12 nodes. For ABySS-map and Novoalign we have $p > 12$. The scalability performance of DIDA for *C. elegans* dataset and ABySS-map alignment method for [2, 4, 8, 16, 32] nodes is presented in Fig. 2.8. For larger datasets such as human genome, the value of p is greater than 12. In general, the larger the datasets

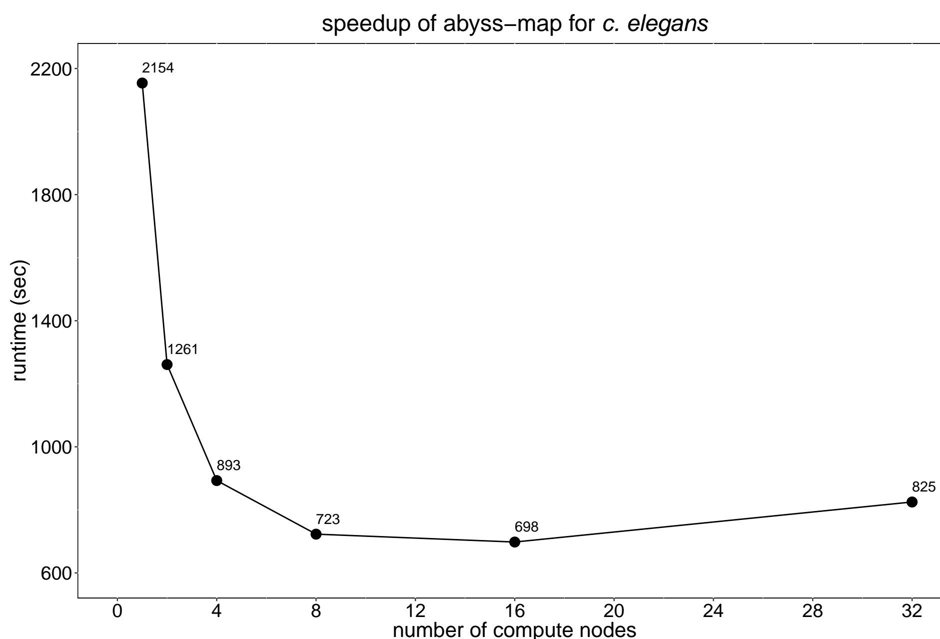

Figure 2.8: The scalability performance of ABySS-map+DIDA for *C. elegans* dataset.

or the slower the alignment methods, the greater the value of p. For instance, the runtime of Bowtie2 within DIDA framework on eight computing nodes is 595 sec compared to 601 sec on 12 computing nodes. This is because of the related overhead of the dispatch and merge steps. Another point that should be explained is the unexpected memory scalability for BWA from 4 to 8 nodes on the *C. elegans* dataset. Based on the size of target set, bwa-index automatically chooses between *bwtsw* and *is* (induced sorting) algorithms to generate BWT in the index construction process. For short target sets ($\leq$ 25Mb), bwa-index uses *is* algorithm while for long target sets ($>$ 25Mb) it employs *bwtsw*. The memory usage of *bwtsw* is less than *is* for a given target set. When we divide *C. elegans* dataset into 8 or more partitions, the size of

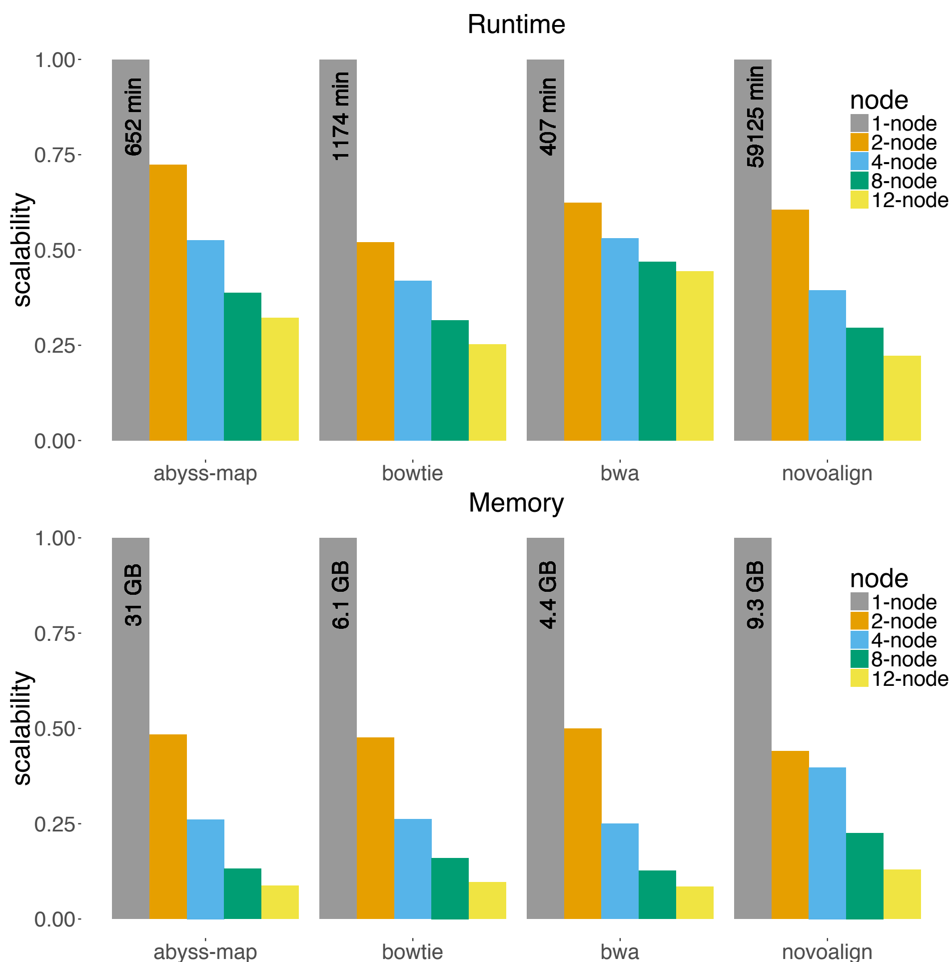

Figure 2.9: Scalability of different aligners using DIDA for human draft assembly [79].

each subset becomes less than 25Mb, and hence, bwa-index automatically invokes *is* algorithm. On the other hand, for 4 partitions or less, bwa-index uses *bwtsw* algorithm. Therefore, we see the memory scalability of BWA for *C. elegans* not behaving as expected.

Fig. 2.9 shows the results on the human draft assembly data. Compared to the smaller datasets, for human genome we see better runtime and

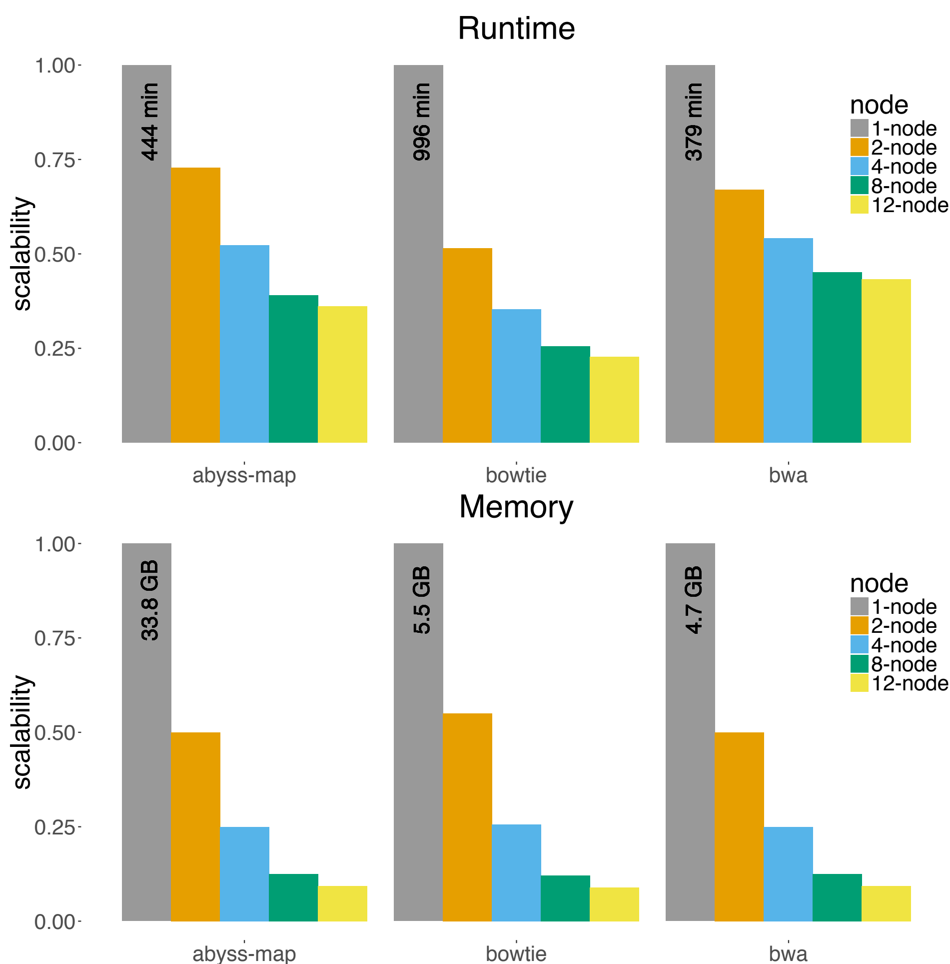

Figure 2.10: Scalability of different aligners using DIDA for hg19.

memory usage scalability, illustrating that DIDA shows better performance on large data due to the amortization of the overhead of the distributed paradigm. That means the overhead of dispatch and merge steps are compensated for large-scale indexing and alignment applications. We have also evaluated the performance of DIDA on the human reference genome (hg19) as a static target set. Fig. 2.10 and Table 2.4 shows the scalability of wall-

clock time and indexing peak memory usage of different aligners, except Novoalign (due to its long runtime). As expected, the scalabilities for runtime and memory are similar to the case of non-static target set.

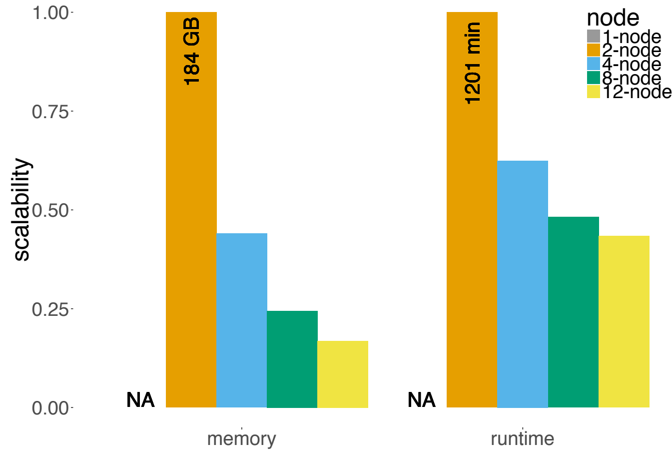

Figure 2.11: Scalability of abyss-map aligner using DIDA for *P. glauca*.

Fig. 2.11 and Table 2.5 shows the result for ABySS-map on *P. glauca* draft assembly. Due to resource restrictions, we could not obtain the result of ABySS-map aligner without DIDA. The required memory for constructing the index for the spruce draft assembly is about 400 GB. However, using DIDA framework we can divide the draft spruce assembly into a number of partitions, and perform the indexing and alignment operation in a distributed way. From Table 2.5, we can easily see the scalability for runtime and indexing peak memory usage of ABySS-map within DIDA.

2.6 Discussion

Indexing large target sequences, and aligning large queries are computationally challenging. In this article, we described a novel, scalable and cost-effective parallel workflow for indexing and alignment, called DIDA.

The performance of DIDA was measured and evaluated when coupled with popular alignment methods BWA, Bowtie2, Novoalign, and ABySS-map on *C. elegans*, human draft genome, human reference genome, and *P. glauca* genome. Compared to their baseline performance, when run through the DIDA framework with 12 nodes, BWA, Bowtie2, Novoalign, and ABySS-map use less memory (91%, 90%, 87%, and 91%, respectively) and execute faster (55%, 74%, 77%, and 67%, respectively) for a draft human genome assembly.

DIDA is an enabling technology for large-scale alignment and assembly tasks especially for labs that have limited compute resources. It enabled our lab at Genome Sciences Centre in BC Cancer Agency to assemble the white spruce (*P. glauca*) genome [8]. For this large-scale *de novo* assembly task, the required memory for index construction from the draft genome on a single node is about 400 GB of RAM, which requires the use of a special big memory machine, and may be prohibitive for many labs. Using the DIDA framework, the indexing was performed in a distributed way on 12 low-memory compute nodes with peak memory usage of 31 GB on any one node. Therefore, DIDA efficiently made this huge indexing and alignment and assembly task feasible, well scalable, and modular [43, 106]. DIDA has also enabled our lab to perform many experiments at a time without

requisitioning large memory machines.

As the cost of DNA sequencing is dropping faster than the cost of computational power, the need for scalable and cost-effective parallel and distributed algorithms and software tools for accurately and expeditiously processing "big data" from high-throughput sequencing is increasing. DIDA offers a solution to this growing issue for the alignment problem, especially when the target is non-static, or large. In life sciences research organizations and clinical genomics laboratories, alignment and *de novo* assembly are becoming two key steps in everyday research and analysis. Especially for labs that have limited computational resources, DIDA may offer an appropriate solution to address their needs and expectations by reducing heavy computational resource requirements.

Chapter 3

ntHash: recursive nucleotide hashing

3.1 Publication note

The work described in this chapter was previously published in the *Bioinformatics* journal in 2016 [77]. The outcome algorithm and software tool from this work was used in three published journal papers in *BMC Medical Genomics* in 2015 [104], *Genome Research* in 2017 [42], and *Bioinformatics* in 2017 [78]. The work described in this chapter is the main work of the author, under the supervision of his PhD supervisor, Inanc Birol. The author designed the algorithm, implemented the software tool, performed the experiments, and wrote the manuscript. The co-authors on this work helped in performing the experiments.

3.2 Author summary

Hashing has been widely used for indexing, querying, and rapid similarity search in many bioinformatics applications, including sequence alignment, genome and transcriptome assembly, k-mer counting, and error correction.

Hence, expediting hashing operations would have a substantial impact in the field, making bioinformatics applications faster and more efficient. In this chapter, we present ntHash, a hashing algorithm tuned for processing DNA/RNA sequences. It performs the best when calculating hash values for adjacent k-mers in an input sequence, operating an order of magnitude faster than the best performing alternatives in typical use cases. ntHash is available online at `http://www.bcgsc.ca/platform/bioinfo/software/nthash` and is free for academic use.

3.3 Introduction

Hashing is a common function across many informatics applications, and refers to mapping an input key value of arbitrary size to an allocated memory of predetermined size. Among other uses, it is an enabling concept for rapid search operations, and forms the backbone of Internet search engines. In bioinformatics, it has many applications including sequence alignment, genome and transcriptome assembly, RNA-seq expression quantification, k-mer counting, and error correction. Large-scale sequence analysis often relies on cataloguing or counting consecutive k-mers in DNA/RNA sequences for indexing, querying, and similarity searching. An efficient way of implementing such operations is through the use of hash based data structures, such as hash tables or Bloom filters. Therefore, improving the performance of hashing algorithms would have a great impact in a wide range of bioinformatics tools. Here, we present ntHash, a fast function for recursively computing hash values for consecutive k-mers.

3.4 Method

We propose an algorithm to hash consecutive k-mers in a sequence, r of length $l > k$, using a recursive function, f, where the hash value of the new k-mer H is computed from the hash value of the previous k-mer:

$$H(k\text{−mer}_i) = f(H(k\text{−mer}_{i-1}), r[i+k-1], r[i-1]) \tag{3.1}$$

Such a recursive function, also called rolling hash function, offers huge improvements in performance when hashing consecutive k-mers. This has been previously described and investigated for n-gram hashing for string matching, text indexing, and information retrieval [17, 32, 45, 51]. In this work, we have customized the concept for hashing all k-mers of a DNA sequence, and implemented an adapted version of the cyclic polynomial hash function, ntHash, to efficiently calculate normal or canonical hash values for k-mers in DNA sequences. In hashing by cyclic polynomial, ntHash uses barrel shifts instead of multiplications to make the process faster. To compute hash values for all k-mers of the sequence r of length l, we first hash the initial k-mer, k-mer$_0$, as follows:

$$H(k\text{−mer}_0) = rol^{k-1}h(r[0]) \oplus rol^{k-2}h(r[1]) \oplus \cdots \oplus h(r[k-1]) \tag{3.2}$$

where rol is a cyclic binary left rotation, $\oplus$ is the bit-wise EXCLUSIVE OR (XOR) operator, and $h(.)$ is a seed table, in which the letters of the DNA alphabet, $\sum = \{A, C, G, T\}$, are assigned different random 64-bit integers. The hash values for all consequent k-mers, k-mer$_1$, ..., k-mer$_{l-k}$, are then

computed recursively as follows:

$$H(k\text{−mer}_i) = rol^1 H(k\text{−mer}_{i-1}) \oplus rol^k h(r[i-1]) \oplus h(r[i+k-1]) \quad (3.3)$$

We note that the time complexity of ntHash for sequence r is $O(k+l)$ compared to $O(kl)$ complexity of regular hash functions. In some bioinformatics applications, one might be interested in computing the hash value of forward and reverse-complement sequences of a k-mer. To do so, we add in the seed table integers that correspond to the complement bases, such that table indices of base-complement base pairs are separated by a fixed offset. Using this table, we can easily compute the hash value for the reverse-complement (as well as the canonical form) of a sequence efficiently, without actually reverse-complementing the input sequence, as follows:

$$\bar{H}(k\text{−mer}_0) = h(r[0]+d) \oplus rol^1 h(r[1]+d) \oplus \cdots \oplus rol^{k-1} h(r[k-1]+d) \quad (3.4)$$

The canonical hash values for all consequent k-mers, k-mer$_1$, ..., k-mer$_{l-k}$, are then computed recursively as follows:

$$\bar{H}(k\text{−mer}_i) = ror^1 \bar{H}(k\text{−mer}_{i-1}) \oplus ror^1 h(r[i-1]+d) \oplus rol^{k-1} h(r[i+k-1]+d) \quad (3.5)$$

where ror is a cyclic binary right rotation, and d is the offset of complement base in the seed table $h(.)$. Additionally, ntHash provides a fast way to calculate multiple hash values for a given k-mer, without iterating the full process for each hash value. To do so, a single hash value is computed from a given k-mer, and then each extra hash value is computed by few more

multiplication, shifting and XOR operations on the initial hash value. This would be very useful for certain bioinformatics applications, such as those that utilize the Bloom filter data structure.

Experimental results show a significant speed improvement over traditional methods, while maintaining a near-ideal hash value distribution. In the Results section, we have used sequencing data on the human individual NA19238 from the Illumina Platinum Genomes project, as well as simulated random DNA sequences.

- Real data of Human individual NA19238 from Illumina Platinum Genome project: `http://sra.dnanexus.com/runs/ERR309932`
- Simulated random DNA sequences using *seqgen* in ntHash package: `https://github.com/bcgsc/ntHash/blob/master/lib/seqgen.hpp`

3.5 Results

A good hash function should generate hash values that are uniformly distributed across the target domain resulting in fewer collisions. To evaluate the uniformity of ntHash, one way is to use the correlation coefficient between the bits of 64-bit hash values. That is, if each bit $x_i, i = 1, \ldots, 64$, in a 64-bit hash value vector $X = \{x_1, x_2, \ldots, x_{64}\}$ is an independent random variable, then there should be no correlation between them. To test this, we first generated sets of hash values on randomly generated DNA sequences. We next computed the correlation coefficient matrix for each sample set, and performed significance tests with Bonferroni correction for the hypothesis that the correlation coefficients are consistent with zero. Fig. 3.4 shows

the correlation coefficients of two sample sets of size 100 (below diagonal) and 100,000 (above diagonal). The plot shows natural statistical fluctuations for smaller sample sets. The correlations dissipate rapidly for large sample sets (Figs. 3.1- 3.5). Comparing the computed correlation coefficients with a confidence interval around the theoretical zero correlation shows that for all hash functions tested, the number of observations outside the 99.7% confidence interval is around 0.3%, in agreement with theoretical expectations..

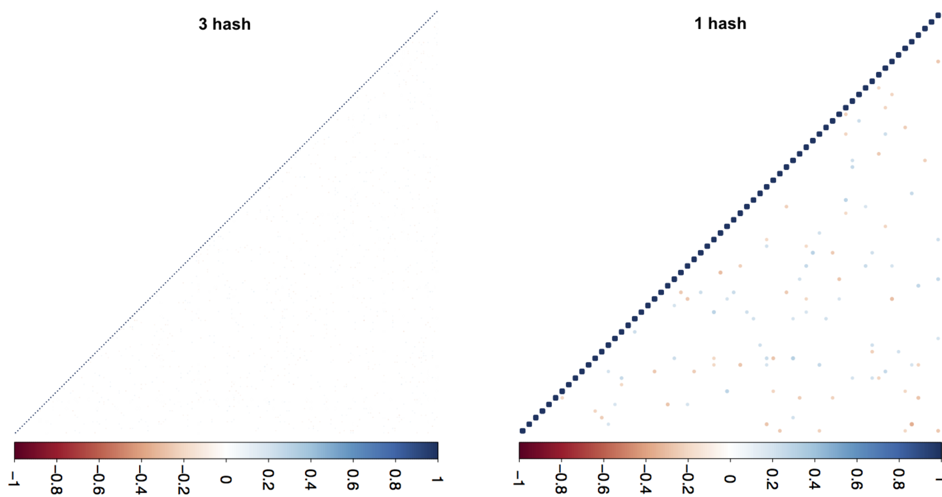

Figure 3.1: Correlation coefficient plots of cityhash for one (right) and three (left) hashes on small (100 data points, below diagonal) and large sample set (100,000 data points, above diagonal).

We have also evaluated the uniformity of different hash methods by utilizing a Bloom filter data structure. We first load a Bloom filter with a number of unique k-mers, and then query the Bloom filter with another set of unique k-mers. The results show the false positive rate of ntHash, as well as other hash functions tested comply with the theoretical false positive rate for Bloom filters, indicating the uniform distribution of hash values

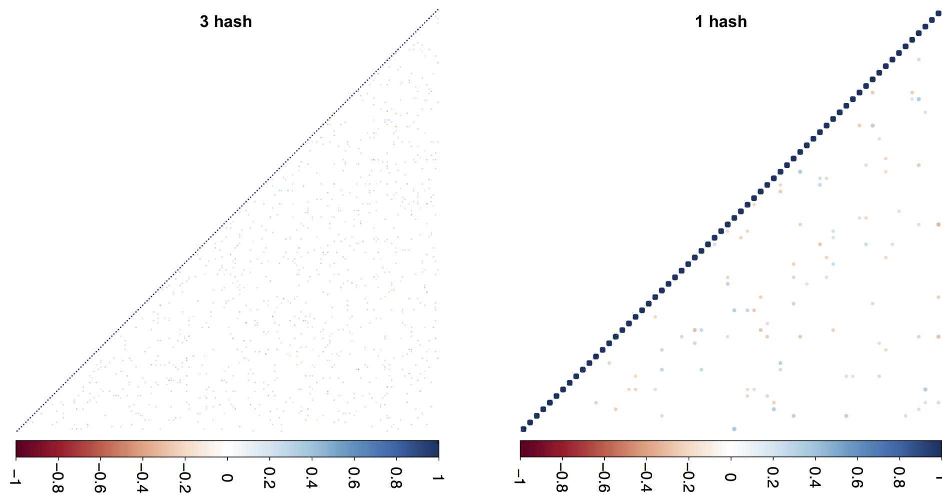

Figure 3.2: Correlation coefficient plots of murmur for one (right) and three (left) hashes on small (100 data points, below diagonal) and large sample set (100,000 data points, above diagonal).

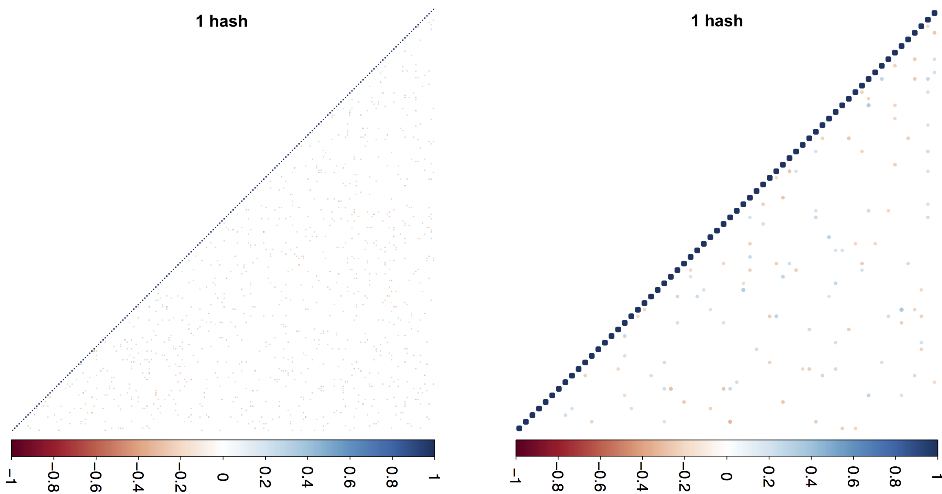

Figure 3.3: Correlation coefficient plots of xxhash for one (right) and three (left) hashes on small (100 data points, below diagonal) and large sample set (100,000 data points, above diagonal).

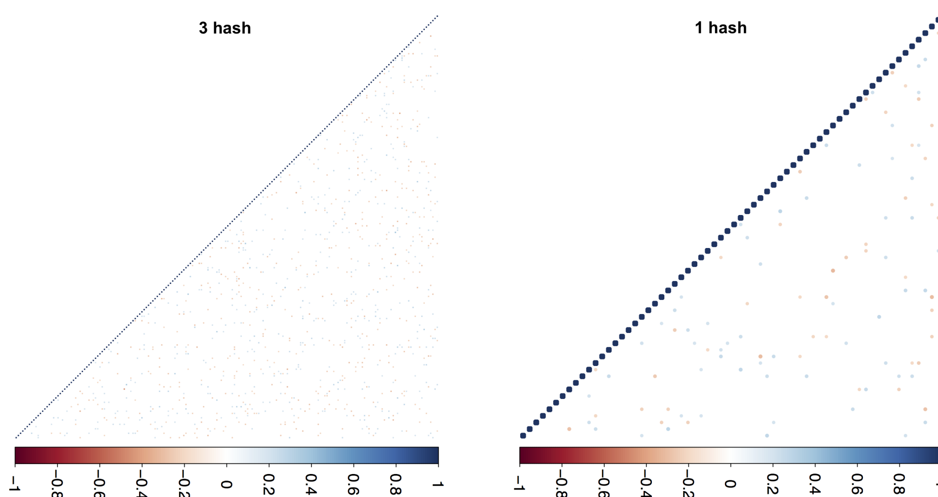

Figure 3.4: Correlation coefficient plots of ntHash for one (right) and three (left) hashes on small (100 data points, below diagonal) and large sample set (100,000 data points, above diagonal).

generated (Table 3.1- 3.8). We have compared the uniformity and runtime performance of ntHash algorithm with three most widely used hash methods in bioinformatics:

- cityhash: `https://github.com/google/cityhash`
- murmur: `https://github.com/aappleby/smhasher`
- xxhash: `https://github.com/Cyan4973/xxHash`

In all experiments, we first load the Bloom filter with 100 long sequences of length 5,000,000bp and allocating 8 bit/k-mer in Bloom filter. Next we query 4,000,000 real reads of length 250bp with k-mer of sizes 50, 150 and 250. The theoretical approximate false positive rate for h=1, 3, and 5 are 11.75%, 3.06% and 2.17%, respectively.

Table 3.1: Bloom filter evaluation for cityhash on real data.

k	Hash#	Set bit#	Query#	False hit#	FPR%
	1	470,017,244	804,000,000	94,378,728	11.74
50	3	1,250,823,778	804,000,000	24,570,368	3.06
	5	1,858,906,097	804,000,000	17,400,323	2.16
	1	469,998,731	404,000,000	47,919,015	11.86
150	3	1,250,821,595	404,000,000	12,338,460	3.05
	5	1,858,931,600	404,000,000	8,748,217	2.17
	1	469,986,185	4,000,000	472,181	11.8
250	3	1,250,802,647	4,000,000	121,776	3.04
	5	1,858,880,965	4,000,000	85,939	2.15

Table 3.2: Bloom filter evaluation for murmur on real data.

k	Hash#	Set bit#	Query#	False hit#	FPR%
	1	470,007,281	804,000,000	94,295,129	11.73
50	3	1,250,852,012	804,000,000	24,560,434	3.05
	5	1,858,955,242	804,000,000	17,381,581	2.16
	1	470,002,596	404,000,000	47,411,326	11.74
150	3	1,250,839,667	404,000,000	12,338,781	3.05
	5	1,858,923,413	404,000,000	8,750,161	2.17
	1	469,997,041	4,000,000	469,325	11.73
250	3	1,250,795,983	4,000,000	122,051	3.05
	5	1,858,914,749	4,000,000	86,649	2.17

Table 3.3: Bloom filter evaluation for xxhash on real data.

k	Hash#	Set bit#	Query#	False hit#	FPR%
	1	470,016,553	804,000,000	94,292,232	11.73
50	3	1,250,840,753	804,000,000	24,518,667	3.05
	5	1,858,942,822	804,000,000	17,394,611	2.16
	1	469,995,918	404,000,000	47,411,924	11.74
150	3	1,250,830,011	404,000,000	12,340,660	3.05
	5	1,858,930,859	404,000,000	8,743,269	2.16
	1	469,986,739	4,000,000	469,547	11.74
250	3	1,250,798,629	4,000,000	122,788	3.07
	5	1,858,874,277	4,000,000	86,494	2.16

Table 3.4: Bloom filter evaluation for ntHash on real data.

k	Hash#	Set bit#	Query#	False hit#	FPR%
	1	469,999,521	804,000,000	94,251,857	11.72
50	3	1,250,842,928	804,000,000	24,535,428	3.05
	5	1,858,926,469	804,000,000	17,420,006	2.17
	1	469,998,307	404,000,000	47,418,185	11.74
150	3	1,250,807,514	404,000,000	12,333,989	3.05
	5	1,858,901,461	404,000,000	8,743,479	2.16
	1	469,989,680	4,000,000	469,293	11.74
250	3	1,250,786,386	4,000,000	122,542	3.06
	5	1,858,855,900	4,000,000	86,775	2.17

Table 3.5: Bloom filter evaluation for cityhash on simulated data.

k	Hash#	Set bit#	Query#	False hit#	FPR%
	1	470,004,731	804,000,000	94,467,037	11.75
50	3	1,250,839,836	804,000,000	24,581,360	3.06
	5	1,858,937,046	804,000,000	17,433,637	2.17
	1	470,001,865	404,000,000	47,475,319	11.75
150	3	1,250,810,005	404,000,000	12,352,225	3.06
	5	1,858,911,301	404,000,000	8,757,714	2.17
	1	469,989,444	4,000,000	469,599	11.74
250	3	1,250,773,973	4,000,000	122,939	3.07
	5	1,858,866,870	4,000,000	87,331	2.18

Table 3.6: Bloom filter evaluation for murmur on simulated data.

k	Hash#	Set bit#	Query#	False hit#	FPR%
	1	470,017,505	804,000,000	94,484,767	11.75
50	3	1,250,827,988	804,000,000	24,589,864	3.06
	5	1,858,925,731	804,000,000	17,424,248	2.17
	1	469,997,827	404,000,000	47,469,314	11.75
150	3	1,250,784,464	404,000,000	12,347,235	3.06
	5	1,858,877,467	404,000,000	8,757,213	2.17
	1	469,992,778	4,000,000	469,332	11.73
250	3	1,250,777,246	4,000,000	121,813	3.05
	5	1,858,869,175	4,000,000	86,187	2.15

Table 3.7: Bloom filter evaluation for xxhash on simulated data.

k	Hash#	Set bit#	Query#	False hit#	FPR%
50	1	470,008,758	804,000,000	94,470,794	11.75
	3	1,250,821,114	804,000,000	24,585,395	3.06
	5	1,858,965,631	804,000,000	17,434,877	2.17
150	1	469,991,012	404,000,000	47,464,074	11.75
	3	1,250,815,185	404,000,000	12,350,539	3.06
	5	1,858,920,189	404,000,000	8,751,526	2.17
250	1	469,997,137	4,000,000	470,919	11.77
	3	1,250,793,131	4,000,000	122,493	3.06
	5	1,858,893,379	4,000,000	86,618	2.17

Table 3.8: Bloom filter evaluation for ntHash on simulated data.

k	Hash#	Set bit#	Query#	False hit#	FPR%
50	1	470,001,008	804,000,000	94,473,191	11.75
	3	1,250,822,696	804,000,000	24,594,765	3.06
	5	1,858,911,305	804,000,000	17,428,958	2.17
150	1	470,009,333	404,000,000	47,464,574	11.75
	3	1,250,812,436	404,000,000	12,351,716	3.06
	5	1,858,915,330	404,000,000	8,757,143	2.17
250	1	469,979,683	4,000,000	469,658	11.74
	3	1,250,795,888	4,000,000	122,625	3.07
	5	1,858,897,893	4,000,000	86,664	2.17

Usually, we compute a single hash value when we hash a given k-mer. However, there are some bioinformatics applications utilizing the Bloom filter data structure that requires computing multiple hash values for a given k-mer. The existing hash methods do this by repeating the whole hashing procedure on a given k-mer with different initial seeds while in ntHash we have provided a version to compute the multiple hash values in an efficient way. Given the number of required hash values, we first compute a hash value for the k-mer in the regular way using NT64 function. We then perturb the 64-bit computed hash values by few additional operations without repeating the whole hashing procedure to get required number of hash values. The detailed procedure for the multi-hash version of ntHash is explained below. The uniformity results for the multi-hash version of ntHash and other hash functions have been presented in Figs. 3.1- 3.4.

Fig. 3.6 shows the runtimes for hashing different length k-mers in 5 million DNA sequences of length 250 bp. In the inset, we see ntHash outperforms other algorithms when hashing more than two consecutive k-mers in a DNA sequence. The runtime of all hash methods for hashing one billion k-mers of lengths $\{50,100,150, 200,250\}$ is shown in Fig. 3.7. We have also calculated the runtime for canonical hashing of 1 billion k-mers presented in Fig. 3.8. From these results, we see ntHash computes the regular and canonical hash values very quickly. Fig. 3.9 illustrates a typical use case of computing multiple hash values for 50-mers in DNA sequences of length 250 bp, and shows that ntHash is over 20$\times$ faster than the closest competitor.

3.6 Discussion

In this work, we introduced ntHash algorithm for computing regular or canonical hash values for all possible k-mers in a DNA/RNA sequence. Being a recursive or rolling hash function, ntHash calculates the hash value for the new k-mer from the hash value of the previous k-mer. This idea boosts the performance of ntHash with significant speed improvement in comparison with conventional hash methods in bioinformatics, and maintains the near-ideal distribution for generated hash values. Moreover, by providing a fast way for computing multi-hash values for a given k-mer, ntHash was demonstrated to be very useful for applications utilizing the Bloom filter data structure. ntHash has been employed in a series of algorithms and software tools to expedite the hashing operations. Examples include our current and new version of ABySS genome assembly software package [42, 98], the new version of BioBloomTools [16], Konnector [103, 104], ChopStitch [46], and our new algorithm for cardinality estimation, ntCard, which will be explained in the next chapter.

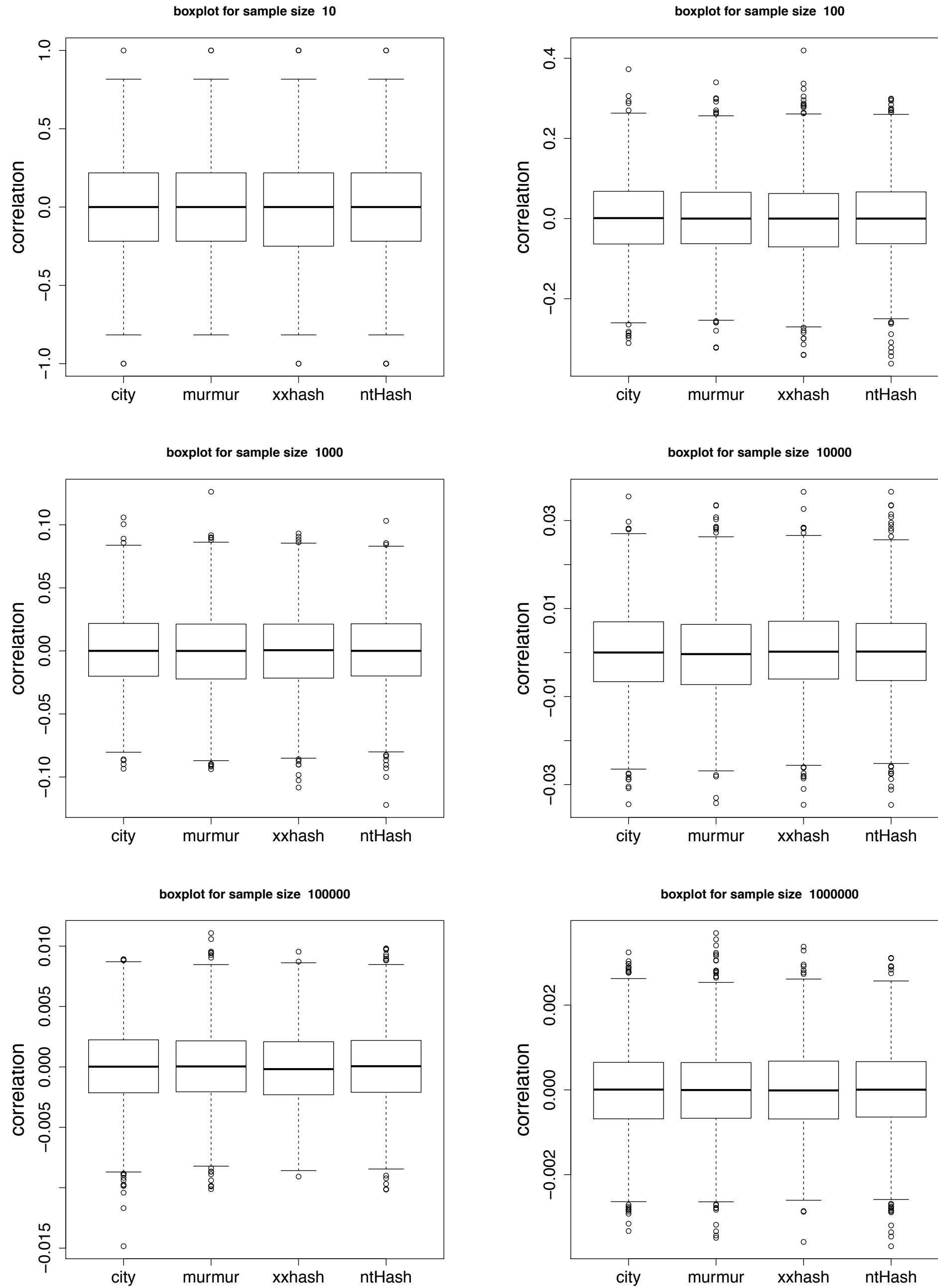

Figure 3.5: Correlation coefficient of hash algorithms for different sample sets. We see the correlation between hash value bits is near-ideal for ntHash as well as the state-of-the-art hashing methods like cityhash, murmur, and xxhash.

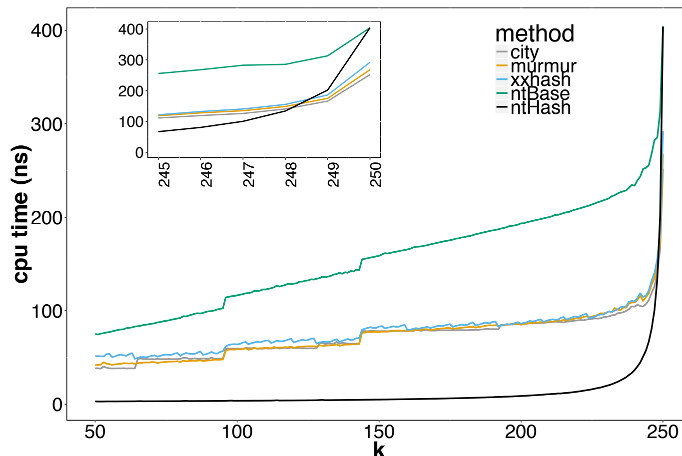

Figure 3.6: Runtime for hashing a 250bp DNA sequence with different k ranging from 50 to 250. ntHash outperforms all other hash methods when hashing more than two k-mers, *i.e.* $k < 249$.

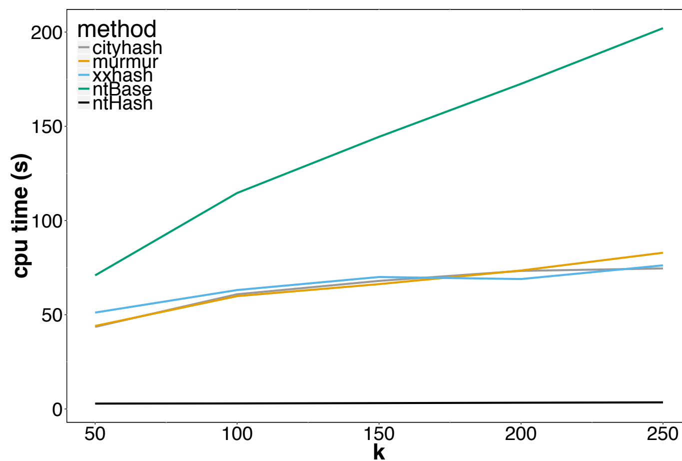

Figure 3.7: CPU time for hashing 1 billion k-mers of lengths 50, 100, 150, 200, and 250. ntBase is the hash function based on non-recursive equation of ntHash.

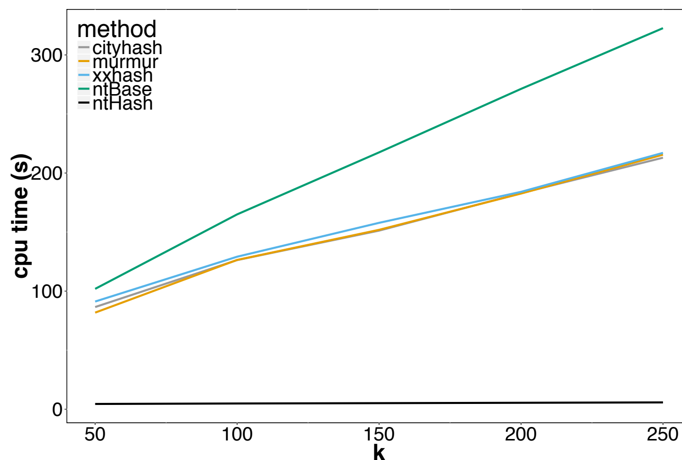

Figure 3.8: CPU time for canonical hashing 1 billion k-mers of lengths 50, 100, 150, 200, and 250. ntBase is the hash function based on non-recursive equation of ntHash.

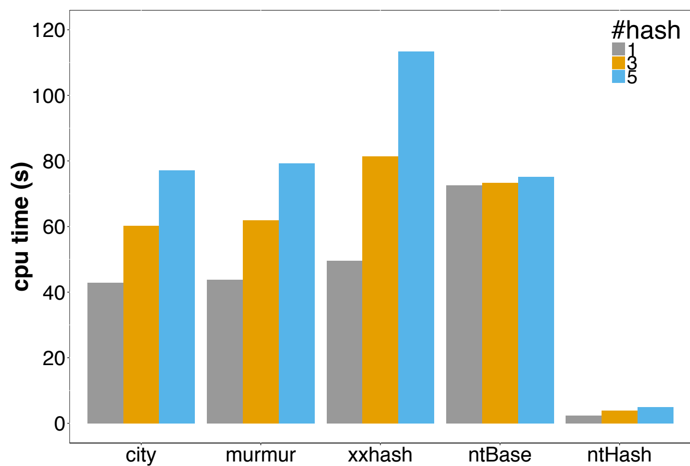

Figure 3.9: Comparing multi-hashing runtime of ntHash with the leading hash functions for one billion 50-mers. ntHash performs over 20 faster than the closest competitor, cityhash. Grey, orange and blue bars refer to calculation of one, three and five hash functions, respectively.

Chapter 4

ntCard: a streaming algorithm for cardinality estimation in genomics data

4.1 Publication note

The work described in this chapter was previously published in the *Bioinformatics* journal in 2017 [78]. The outcome algorithm and software tool from this work was used in one published journal paper in *Genome Research* in 2017 [42], and one submitted journal paper in *Bioinformatics* in 2017 [46]. The work described in this chapter is the main work of the author, under the supervision of his PhD supervisor, Inanc Birol. The author designed the algorithm, implemented the software tool, performed the experiments, and wrote the manuscript. The co-author on this work, Hamza Khan, helped in performing the experiments.

4.2 Author summary

Many bioinformatics algorithms are designed for the analysis of sequences of some uniform length, conventionally referred to as k-mers. These include de Bruijn graph assembly methods and sequence alignment tools. An efficient algorithm to enumerate the number of unique k-mers, or even better, to build a histogram of k-mer frequencies would be desirable for these tools and their downstream analysis pipelines. Among other applications, estimated frequencies can be used to predict genome sizes, measure sequencing error rates, and tune runtime parameters for analysis tools. However, calculating a k-mer histogram from large volumes of sequencing data is a challenging task. In this chapter, we present ntCard, a streaming algorithm for estimating the frequencies of k-mers in genomics datasets. At its core, ntCard uses the ntHash algorithm to efficiently compute hash values for streamed sequences. It then samples the calculated hash values to build a reduced representation multiplicity table describing the sample distribution. Finally, it uses a statistical model to reconstruct the population distribution from the sample distribution. Our benchmarks demonstrate ntCard as a potentially enabling technology for large-scale genomics applications. ntCard is implemented in C++ and is released under the GPL-3 license. The software and source codes are freely available at `https://github.com/bcgsc/ntCard`.

4.3 Introduction

Many bioinformatics applications rely on counting or cataloguing fixed-length substrings of DNA/RNA sequences, called k-mers, generated from

reads coming out of high-throughput sequencing platforms. This is a very important step in *de novo* assembly [11, 61, 91, 98, 109], multiple sequence alignment [22], error correction [35, 72], repeat detection [96], SNP detection [82, 94], and RNA-seq quantification analysis [85]. The problem of counting k-mers has been well studied in the literature, including the Jellyfish [71], BFCounter [74], DSK [88], and KMC [21] algorithms. These tools need considerable computational resources and can be improved in terms of memory, disk space, and runtime requirements for processing and obtaining the histogram of k-mer frequencies in large sets of DNA/RNA sequences. During the past few years there have been many studies to improve the memory and time requirements for the k-mer counting problem. While a naïve approach would keep track of all possible k-mers in the input datasets, employing a succinct and compact data structure [18], or a disk-based workflow [21, 88] would reduce memory usage. Although the improved methods with efficient implementations have considerable impact on memory and time usage, they require processing of all possible k-mers base-by-base and storing them in memory or disk. Therefore, the time and memory requirements for theses efficient solutions grow linearly with the input data size, and can take hours or days using terabytes of memory for large datasets. In the recent works by Chikhi-Medvedev [13] and Melsted-Halldrsson [75], the authors proposed methods to approximate the k-mer coverage histogram in large sets of DNA/RNA sequences, which are about an order of magnitude faster, and require only a small portion of the memory compared with previous k-mer counting algorithms [75]. However, these methods still can take considerable amount of time for processing terabytes

of high-throughput sequencing data, or may not provide the full histogram for k-mer abundance.

In this article, we present an efficient streaming algorithm, *ntCard*, for estimating the k-mer coverage histogram for large high-throughput sequencing genomics data. The proposed method requires fixed amount of memory, and runs in linear time with respect to the size of the input dataset. At its core, ntCard uses the *ntHash* algorithm [77] to efficiently compute hash values for streamed sequences. It samples the calculated hash values to build a reduced representation multiplicity table describing the sample distribution, which it uses to statistically infer the population distribution. We compare the histograms estimated by ntCard with the exact k-mer counts of DSK [88], and illustrate that the ntCard estimations are approximations within guaranteed intervals. We also compare the accuracy, runtime and memory usage of ntCard with the best available exact and approximate algorithms for k-mer count frequencies such as DSK [88], KmerGenie [13], KmerStream [75], and Khmer [41].

4.4 Method

Let's first introduce the problem background and notations on streaming algorithms for identifying the distinct elements. Then we will derive a statistical model to estimate k-mer frequencies, and outline the generated model.

4.4.1 Background, notations, and definitions

Streaming algorithms are algorithms for processing data that are too large to be stored in available memory, but can be examined online, typically in a single pass. There has been a growing interest in streaming algorithms in a wide range of applications, in different domains dealing with massive amounts of data. Examples include, analysis of network traffic, database transactions, sensor networks, and satellite data feeds [19, 20, 40].

Here, we propose a streaming algorithm to estimate the frequencies of k-mers in massive data produced from high-throughput sequencing technologies. Let f_i denote the number of distinct k-mers that occur i times in a given sequencing dataset. The k-mer frequency histogram is then the list of f_i, $i \geq 1$. The kth frequency moment F_k is defined as

$$F_k = \sum_{i=1}^{\infty} i^k . f_i \tag{4.1}$$

The numbers F_k provide useful statistics on the input sequences. For example, F_0 denotes the number of distinct k-mers appearing in the stream sequences, F_1 is the total number of k-mers in the input datasets, F_2 is the Gini index of variation that can be used to show the diversity of k-mers, and F_∞ is related to the most frequent k-mer in the input reads.

There are streaming algorithms in the literature for estimating different kth frequency moments. The problem of estimating F_0, also known as distinct elements counting, has been addressed by the FM-Sketch [28] and K-Minimum Value [4] algorithms. An F_2 estimation algorithm was first proposed in Alon *et al.* [2], and F_∞ was investigated by Cormode and

Muthukrishnan [20]. These proposed algorithms can perform their estimations within a factor of $(1 \pm \epsilon)$ with a set probability using $O(\epsilon^{-2}\log(D))$ operations, where D is the number of distinct k-mers in the dataset [75].

4.4.2 Estimating k-mer frequencies, f_i

To estimate the k-mer frequencies, we use a hash-based approach similar to the KmerStream algorithm [75]. KmerStream is based on the K-Minimum Value algorithm [4], and it samples the data streams at different rates to select the optimal sampling rate giving the best result.

ntCard works by first hashing the k-mers in read streams, which it samples to build a reduced multiplicity table. After calculating the multiplicity table for sampled k-mers, it uses this table to infer the population histogram through a statistical model.

Hashing

ntCard utilizes the ntHash algorithm [77] to efficiently compute the canonical hash values for all k-mers in DNA sequences. ntHash is a recursive, or rolling, hash function in which the hash value for the next k-mer in an input sequence of length l $(l \geq k)$ is derived from the hash value of the previous k-mer as described in the previous chapter.

We have shown earlier that ntHash has significant speed improvement over conventional approaches, while maintaining a near-ideal hash value distribution [77].

Figure 4.1: 64-bit hash value generated by ntHash. The s left bits are used for sampling the k-mers in input datasets and the r right bits are used as resolution bit for building the reduced multiplicity table, with $r + s < 64$.

Sampling and building the multiplicity table

After computing the hash values for k-mers in DNA streams using ntHash, ntCard segments the 64-bit hash values into three parts as shown in Figure 4.1. It uses the left s bits in the 64-bit hash value for its *sampling* criterion, picking k-mers for which these bits are zero, resulting in an average sampling rate of $1/2^s$. Earlier, we have demonstrated that ntHash bits are independently and uniformly distributed [77]. Consequently, on average 1/2 of the hash values start with 0, 1/4 of them will start with two zeros, and $1/2^s$ will start with s zeros. Therefore, by selecting the hash values starting with s zeros, we build our sample with the cardinality of $1/2^s$.

Also building on the statistical properties of computed hash values, we use the right r bits, called the *resolution* bits, to build a k-mer multiplicity table for sampled k-mers. To do so, we use an array of size 2^r to keep observed k-mer counts. The resolution bits of each hash value serve as the index for the count array. We note that, each entry in the array is an approximate count of the sampled k-mers, since there may be multiple k-mers with the same r bit pattern, resulting in count collisions.

Ideally, one would want a hash function that generates a unique hash value for every k-mer, say using infinite number of bits. Also, if one has access to infinite memory to hold all these values, the ideal values for s and

r would be zero and infinity, respectively. Since we do not in practice have access to such resources, we use 64-bit hash values, subsample our dataset by $1/2^s$, and tabulate 2^r patterns (some with zero counts). To infer the population histogram from these measurements, we derived the following statistical model.

Let's denote the count array with 2^r entries by $t^{(r)}$. If we were to extend our resolution to $r + 1$, we would obtain a new count array, $t^{(r+1)}$, with 2^{r+1} entries, twice the size of the current array $t^{(r)}$. There is a relation between the entries of the current array $t^{(r)}$ and the new count array $t^{(r+1)}$. By folding the first half of $t^{(r+1)}$ with its second half, we can construct $t^{(r)}$ using

$$t_n^{(r)} = t_n^{(r+1)} + t_{2^r+n}^{(r+1)}, \qquad \forall n \in [0, \dots, 2^r - 1] \tag{4.2}$$

where $t_n^{(r)}$ denotes the count for entry n in the table $t^{(r)}$.

Next, if we let $p_i^{(r)}$ be the relative frequency of counts $i \geq 0$ in table $t^{(r)}$, with $\sum_{i=0}^{\infty} p_i^{(r)} = 1$, we can make the following observations. An entry of $t_i^{(r)} = 0$ is only possible if $t_i^{(r+1)} = 0$ and $t_{2^r+i}^{(r+1)} = 0$. Since there is no *a priori* reason why the first and second half of $t^{(r+1)}$ should have different count distributions, we can relate the frequencies of zero counts in the two tables through

$$p_0^{(r)} = (p_0^{(r+1)})^2 \tag{4.3}$$

Similarly, a count of one in $t^{(r)}$ is only possible if the first half of $t^{(r+1)}$ is a one and the second half a zero corresponding to that entry, or vice versa,

which we can write mathematically as

$$p_1^{(r)} = 2p_0^{(r+1)} p_1^{(r+1)} \tag{4.4}$$

This can be generalized as

$$p_i^{(r)} = \sum_{i'=0}^{i} p_{i'}^{(r+1)} p_{i-i'}^{(r+1)} \tag{4.5}$$

Note that, equations (4.3) - (4.5) can be solved for $p_i^{(r+1)}$ through the recursive formula

$$p_i^{(r+1)} = \begin{cases} \left(p_0^{(r)}\right)^{1/2} & \text{for } i = 0 \\ \frac{p_1^{(r)}}{2p_0^{(r+1)}} & \text{for } i = 1 \\ \frac{1}{2p_0^{(r+1)}} \left(p_i^{(r)} - \sum_{i'=1}^{i-1} p_{i'}^{(r+1)} p_{i-i'}^{(r+1)}\right) & \text{for } i > 1 \end{cases} \tag{4.6}$$

Now, just like extensions from a resolution of r to $r+1$, resolution to $r+x$ is also mathematically tractable. Ultimately, we would be interested in relating the observed count frequencies $p_i^{(r)}$ to the count frequencies $p_i^{(\infty)}$, and in calculating k-mer multiplicity frequencies

$$\hat{f}_i = \frac{p_i^{(\infty)}}{1 - p_0^{(\infty)}} \tag{4.7}$$

For example, for $i = 1$, this can be calculated as

$$\hat{f}_1 = \lim_{x \to \infty} \frac{\frac{p_1^{(r)}}{2^x (p_0^{(r)})^{\frac{2^x - 1}{2^x}}}}{1 - (p_0^{(r)})^{\frac{1}{2^x}}} = \frac{-p_1^{(r)}}{p_0^{(r)} \ln p_0^{(r)}} \tag{4.8}$$

and for $i = 2$ as

$$\hat{f}_2 = \frac{-p_0^{(r)} p_2^{(r)} + \frac{1}{2}(p_1^{(r)})^2}{(p_0^{(r)})^2 \ln p_0^{(r)}} \tag{4.9}$$

In general, for $\hat{f}_i, i \geq 1$, we can write the following equation

$$\hat{f}_i = \frac{1}{(p_0^{(r)})^i \ln p_0^{(r)}} \sum_{j=0}^{i-1} \frac{(-1)^{i+j}(p_0^{(r)})^j}{i-j} \Bigg(\sum_{\substack{\forall (l,u)\in\mathbb{Z}^2 \text{s.t.} \\ \sum_k u_k = i-j \\ \sum_k l_k u_k = i}} \prod_{k=1}^{|u|} \binom{i-j-\sum_{k'=0}^{k-1} u_{k'}}{u_k} (p_{l_k}^{(r)})^{u_k} \Bigg) \tag{4.10}$$

where $u_0 = 0$, $u_k \neq u_{k'}$ for all $k \neq k'$, and $|u| = \underset{k}{\operatorname{argmax}}\{u_k\}$.

This complex-looking formula can also be written in the following recursive form

$$\hat{f}_i = \frac{-p_i^{(r)}}{p_0^{(r)} \ln p_0^{(r)}} - \frac{1}{i} \sum_{j=1}^{i-1} \frac{j p_{i-j}^{(r)} \hat{f}_j}{p_0^{(r)}} \tag{4.11}$$

The two terms of this equation can be interpreted as follows. The first term corresponds to count frequencies i in table $t^{(r)}$ assuming none of the entries collided with any non-zero entries through folding rounds from $\lim_{x\to\infty}(r+x)$ to r. The second term is a correction to the first term, accounting for all collisions of $(i-j), 0 < j < i$ and j, result of which is a count frequency of i.

Now, we can derive an estimate for F_0 by a similar approach we used for relative frequencies.

$$F_0 = \lim_{x\to\infty} 2^s (1 - p_0^{(r+x)}) 2^{r+x} \tag{4.12}$$

This formula has three terms inside the limit, the first one, 2^s, correcting for the subsampling we have performed. The second term is the frequency of non-zero entries in table $t^{(r+x)}$, and the third entry is the normalizing factor that was used to convert occurrences of counts in this table to their frequencies, $p_i^{(r+x)}$. Taking this limit then gives

$$F_0 = -2^{s+r} \ln p_0^{(r)} \tag{4.13}$$

Using the equations (4.11) and (4.13) we can obtain the k-mer coverage frequencies as outlined in Algorithm 4.1 with a binomial proportion confidence interval. The workflow of ntCard algorithm is also presented in Fig. 4.2.

Program 4.1 The ntCard algorithm

```
1:  function UPDATE(k-mer)
2:      for each read seq do
3:          for each k-mer in seq do
4:              h ← ntHash (k-mer)              ▷ Compute 64-bit h using ntHash
5:              if h_{64:64-s+1} = 0^s then     ▷ Checking the s left bit in h
6:                  i ← h_{r:1}                 ▷ r is resolution parameter
7:                  t_i ← t_i + 1
8:  function ESTIMATE
9:      for i ← 1 to 2^r do
10:         p_{t[i]} ← p_{t[i]} + 1
11:     for i ← 1 to t_max do
12:         p_i ← p_i/2^r
13:     F_0 = −ln p_0 × 2^{s+r}                 ▷ F_0 estimate
14:     for i ← 1 to t_max do
15:         f̂_i ← −p_i/(p_0 ln p_0) − (1/i) Σ_{j=1}^{i−1} (j p_{i−j} f̂_j)/p_0   ▷ Relative frequency estimates
16:     for i ← 1 to t_max do
17:         f_i ← f̂_i × F_0                     ▷ f_i estimates
18:     return f, F_0
```

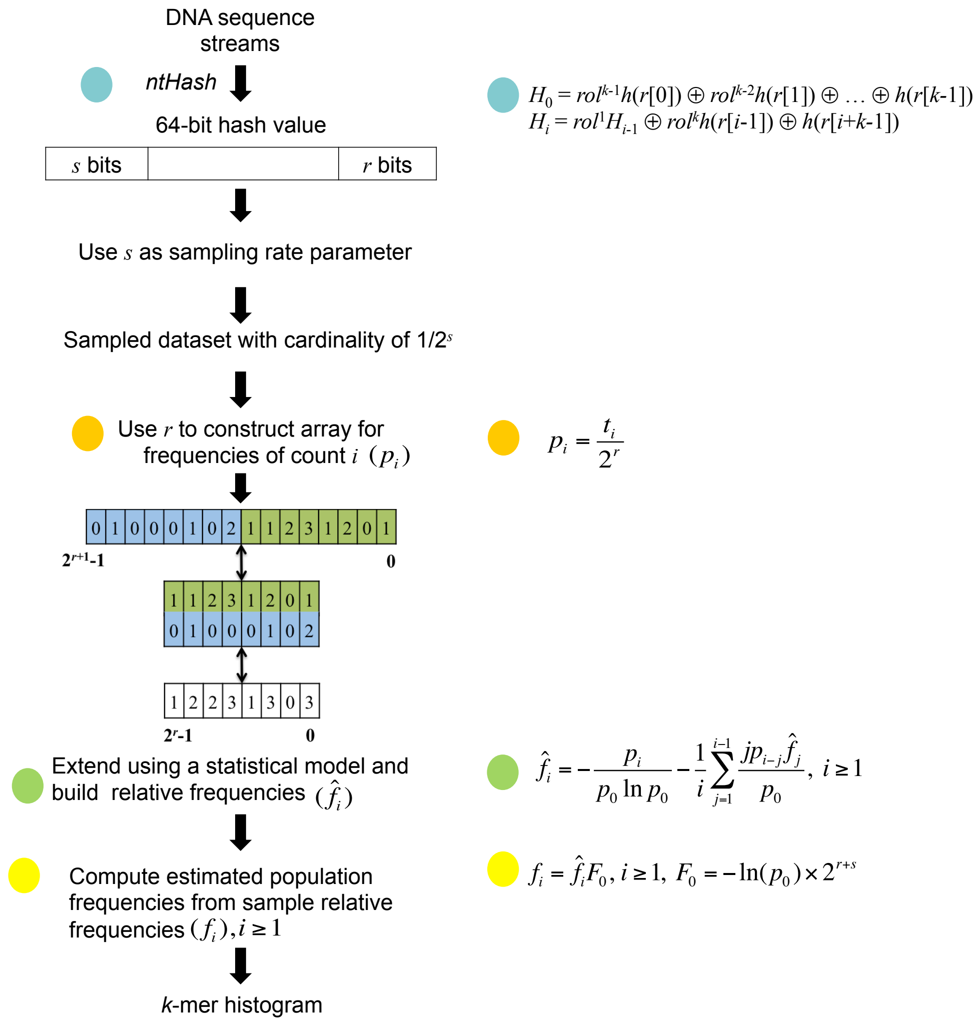

Figure 4.2: The workflow of ntCard algorithm for estimating the k-mer coverage frequencies and the total number of distinct k-mers in DNA sequence streams.

4.4.3 Implementation details

Selection of the resolution parameter, r, represents a tradeoff between accuracy and computational resources. While it should not be too low to avoid poor estimates of frequency counts, it should not be too high for feasible peak memory usage. In our experience, values $r > 20$ work well for accurate estimates, and the memory usage peaks above 1 GB for $r \geq 28$. We have set the default value to $r = 27$. We have also observed that estimations based on only t_r, without applying the statistical model, has higher error rates due to count collisions, as expected.

If input reads or sequences contain ambiguous bases, or characters other than $\{A, C, G, T\}$, ntCard ignores them in the hashing stage. This is performed as a functionality of ntHash algorithm. When ntHash encounters a non-$ACGT$ character it can jump over the ambiguous base, and restarts the hashing procedure from the first valid k-mer containing only $ACGT$ characters.

ntCard is implemented in C++ and parallelized for multi-threading on a single compute node by OpenMP. As input, it gets the set of sequences in FASTA, FASTQ, SAM, or BAM formats. The input sequences can also be in compressed formats such as .gz and .bz formats. ntCard is distributed and released under GNU General Public License (GPL-3). Documentation, software, and source codes are all freely available at `https://github.com/bcgsc/ntCard`.

4.5 Results

4.5.1 Experimental setup

To evaluate the performance and accuracy of ntCard, we downloaded the following publicly available sequencing data.

- The Genome in a Bottle (GIAB) project [110] sequenced seven individuals using a large variety of sequencing technologies. We downloaded 2x250 bp paired-end Illumina whole genome shotgun sequencing data for the Ashkenazi mother (HG004).
- We downloaded a second *H. Sapiens* dataset from the 1000 Genomes Project, for the individual NA19238 (SRA:ERR309932).
- To represent a larger problem, we used the white spruce (*Picea glauca*) genome sequencing data that represents the genotype PG29 [106] (accession number: ALWZ0100000000 and PID: PRJNA83435).

The information of each dataset including the number of sequences, size of sequences, total number of bases, and total input size of datasets is presented in Table 1. To evaluate the performance of ntCard, we compare it to KmerGenie, KmerStream, and Khmer in terms of accuracy of estimates, runtime, and memory usage. We also compare the accuracy of our results with DSK, which is an exact k-mer counting tool. Results were obtained on computing nodes with 48 GB of RAM and dual Intel Xeon X5650 2.66GHz CPUs with 12 cores. The operating system on each node was Linux CentOS 5.4.

Table 4.1: Dataset specification.

Dataset	Read number	Read length	Total bases	Size
HG004	868,593,056	250 bp	217,148,264,000	480 GB
NA19238	913,959,800	250 bp	228,489,950,000	500 GB
PG29	6,858,517,737	250 bp	1,714,629,434,250	2.4 TB

All five tools are run with their default parameters, and the parameters related to the resource usage are set in a way to utilize the maximum capacity on each computing node. For example, all tools are run in multi-threaded mode with the maximum number of threads available on the computer.

4.5.2 Accuracy

In Tables 4.2-4.4, we see the results of DSK, ntCard, KmerGenie, KmerStream, and Khmer for distinct number of k-mers, F_0, as well as the number on singletons, f_1, on three datasets. We compared the accuracy of estimated counts from ntCard, KmerGenie, KmerStream, and Khmer with exact counts from DSK. We see that, for all k-mer lengths, ntCard computes F_0 and f_1 for all three datasets with error rates less than 0.7%. In comparison, the error rates of KmerGenie, KmerStream, and Khmer can be up to 17%, 9%, and 11%, respectively. Note that, the Khmer algorithm only estimates the total number of distinct k-mers, F_0.

Compared to ntCard and KmerStream, Khmer and KmerGenie estimates for distinct number of k-mers, F_0, have the highest error rates (>7%) on PG29 data; though, for HG004 and NA19238, Khmer estimates F_0 with lower error rates, and KmerGenie has very accurate estimates with error rates <1% for all k values. On all three datasets, KmerStream has more

Table 4.2: Accuracy of algorithms in estimating F_0 and f_1 for HG004 reads. The DSK column reports the exact k-mer counts, and columns for the for other tools report percent errors.

k		DSK	ntCard	KmerGenie	KmerStream	Khmer
32	f_1	13,319,957,567	**0.01%**	0.97%	7.04%	–
	F_0	16,539,753,749	**0.02%**	0.64%	5.12%	0.67%
64	f_1	17,898,672,342	**0.02%**	0.35%	0.73%	–
	F_0	21,343,659,785	**0.00%**	0.22%	0.66%	0.15%
96	f_1	18,827,062,018	0.36%	0.87%	**0.00%**	–
	F_0	22,313,944,415	0.24%	0.69%	**0.05%**	0.31%
128	f_1	18,091,241,186	**0.36%**	0.76%	0.40%	–
	F_0	21,555,678,676	0.25%	0.62%	**0.20%**	0.30%

Table 4.3: Accuracy of algorithms in estimating F_0 and f_1 for NA19238 reads. The DSK column reports the exact k-mer counts, and columns for the for other tools report percent errors.

k		DSK	ntCard	KmerGenie	KmerStream	Khmer
32	f_1	14,881,561,565	**0.00%**	0.53%	6.36%	–
	F_0	18,091,801,391	**0.00%**	0.40%	4.64%	1.82%
64	f_1	19,074,667,480	**0.02%**	0.75%	0.68%	–
	F_0	22,527,419,136	**0.01%**	0.77%	0.65%	1.22%
96	f_1	19,420,503,673	0.22%	0.66%	**0.09%**	–
	F_0	22,932,238,161	0.16%	0.66%	**0.07%**	0.46%
128	f_1	17,902,027,438	0.21%	0.85%	**0.19%**	–
	F_0	21,421,517,759	0.13%	0.76%	**0.03%**	1.05%

Table 4.4: Accuracy of algorithms in estimating F_0 and f_1 for PG29 reads. The DSK column reports the exact k-mer counts, and columns for the for other tools report percent errors.

k		DSK	ntCard	KmerGenie	KmerStream	Khmer
32	f_1	27,430,910,938	**0.02%**	15.33%	9.41%	–
	F_0	42,642,198,777	**0.01%**	11.02%	7.37%	8.86%
64	f_1	44,344,130,469	**0.04%**	16.36%	2.61%	–
	F_0	67,800,291,613	**0.02%**	11.14%	1.73%	11.18%
96	f_1	43,300,244,443	**0.66%**	17.51%	0.73%	–
	F_0	69,855,690,006	**0.46%**	11.13%	0.57%	9.36%
128	f_1	32,089,613,024	0.40%	14.82%	**0.06%**	–
	F_0	58,195,246,941	0.30%	8.35%	**0.27%**	7.39%

accurate estimates for longer k-mers, where error rates increase rapidly for shorter k-mers. Although ntCard generally has the opposite trend, it also has the most stable performance for all three datasets. Except for k=128 bp on NA19238 and PG29, and k=96 bp on NA19238 and HG004, ntCard consistently displays the best accuracy both for F_0 and f_1, as indicated by the bold entries in Tables 4.2-4.4, and in all these cases it was a close second.

We have also evaluated the accuracy of full k-mer frequency histograms of ntCard on all three datasets with different k values. Since the Kmer-Stream algorithm only computes estimates for F_0 and f_1 and Khmer only estimates F_0, we could only compare the accuracy of the ntCard histogram with the estimated results of KmerGenie and the exact histogram from DSK method. Figures 4.3-4.5 shows the k-mer frequency histograms of DSK, ntCard, and KmerGenie for all three datasets with four k values, $\{32, 64, 96, 128\}$. Since the results of f_1 have already been presented in Tables 4.2-4.4, and because $f_2..f_{62} \ll f_1$, the histograms in Figures 4.3-4.5 show the k-mer frequencies starting from f_2. From Figures 4.3-4.5 and Tables 4.2-4.4, we can see ntCard estimates the k-mer frequency histograms for all three datasets more accurate than KmerGenie.

4.5.3 Runtime and memory usage

We have calculated the memory usage of all benchmarked tools. DSK uses both main memory and disk space for counting k-mers, and therefore we obtained both values for it. We should also mention that DSK was executed on compute nodes equipped with solid-state drives (SSD). This helps the runtime of DSK be greatly reduced with the SSD and multi-threaded par-

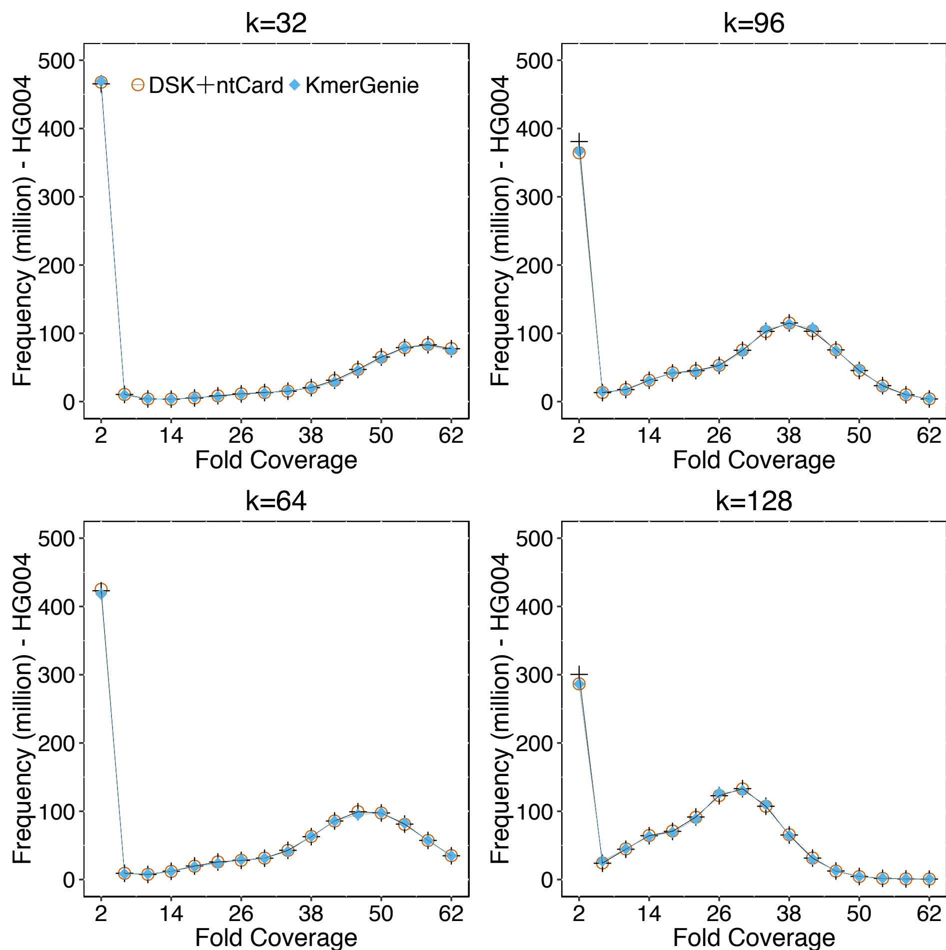

Figure 4.3: k-mer frequency histograms for human genome HG004. We have used DSK k-mer counting results as our ground truth in evaluation (orange circle data points). The k-mer coverage frequency results, $f_2..f_{62}$ of ntCard and KmerGenie for different values of $k = 32, 64, 96, 128$ (the four columns from left to right) are shown with the symbols $(+)$ and $(\diamond)$, respectively.

allelism. The memory usage for DSK on all three datasets was the same at about 20 GB of RAM, while the disk space usage was 500 GB for human genomes HG004 and NA19238, and 1 TB for the white spruce genome PG29.

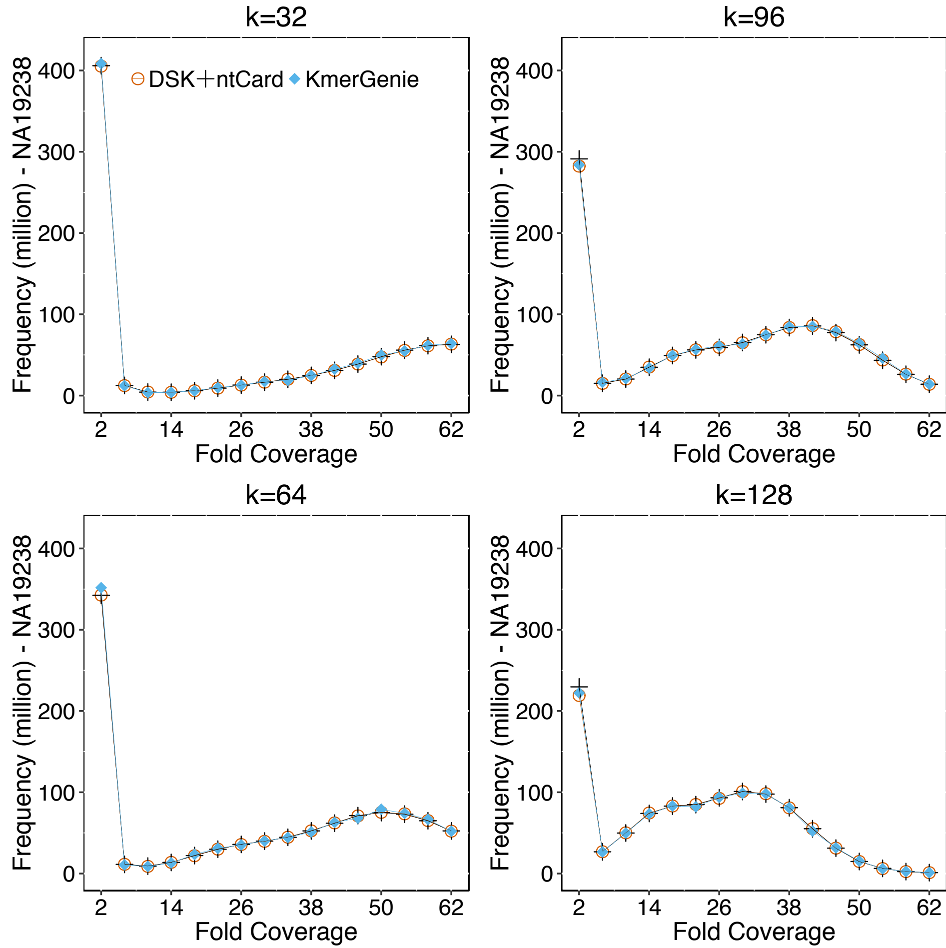

Figure 4.4: k-mer frequency histograms for human genome NA19238. We have used DSK k-mer counting results as our ground truth in evaluation (orange circle data points). The k-mer coverage frequency results, $f_2..f_{62}$ of ntCard and KmerGenie for different values of $k = 32, 64, 96, 128$ (the four columns from left to right) are shown with the symbols $(+)$ and $(\diamond)$, respectively.

The memory usage of KmerGenie to estimate the full k-mer frequency histograms for all datasets was about 200 MB of RAM. KmerStream uses 2-bit counters to estimate F_0 and f_1, resulting in lower memory requirement.

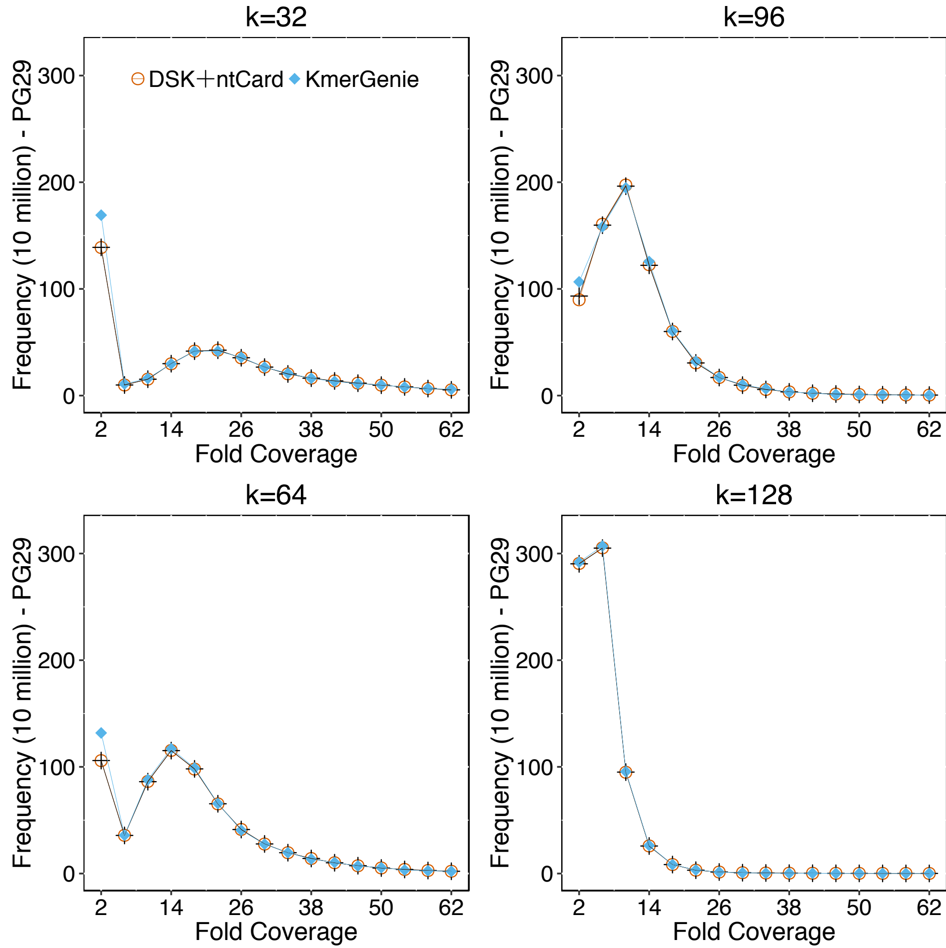

Figure 4.5: k-mer frequency histograms for the white spruce genome PG29. We have used DSK k-mer counting results as our ground truth in evaluation (orange circle data points). The k-mer coverage frequency results, $f_2..f_{62}$ of ntCard and KmerGenie for different values of $k = 32, 64, 96, 128$ (the four columns from left to right) are shown with the symbols $(+)$ and $(\diamond)$, respectively.

The memory usage for KmerStream on all three datasets was about 65 MB of RAM. The Khmer algorithm requires the lowest amount of memory among all algorithms but only estimates F_0. It requires about 15 MB of

RAM to estimate the total number of distinct k-mers in all three datasets. The memory requirement of ntCard for all three datasets was about 500 MB of RAM, although we note that it computes the full k-mer multiplicity histogram. We have also implemented a special runtime parameter to only compute the total number of distinct elements, F_0, in which case it requires about 2 MB of RAM.

Figures 4.6-4.8 shows the runtime of all methods on the experimented datasets with different k values from 32 to 128. The runtime of ntCard to obtain the full k-mer frequency histograms for human genome datasets (HG004, NA19238) is about 6 mins. For KmerStream, it takes about 100 mins to obtain F_0 and f_1 on human genome datasets, while this is about 200 mins for Khmer to estimate just the total number of distinct k-mers, F_0. DSK and KmerGenie take up to 600 and 800 minutes, respectively, to compute the k-mer coverage histograms for human genome datasets. For the white spruce PG29 dataset, ntCard requires about 30 mins to estimate k-mer frequency histograms, while for KmerStream it takes about 450 mins to obtain F_0 and f_1. The Khmer takes longer about 1200 mins to estimate F_0. DSK and KmerGenie can take up to 2700 and 3400 mins to compute the k-mer frequency histograms.

We should note that ntCard, KmerGenie, and KmerStream algorithms have an option to pass multiple k values and compute multiple k-mer coverage histograms in a single run. This option will reduce the amortized runtime per k value, but it will increase the memory usage. From the runtime results, we see ntCard estimates the full k-mer coverage frequency histograms $> 15\times$ faster than the closest competitor, KmerStream, which

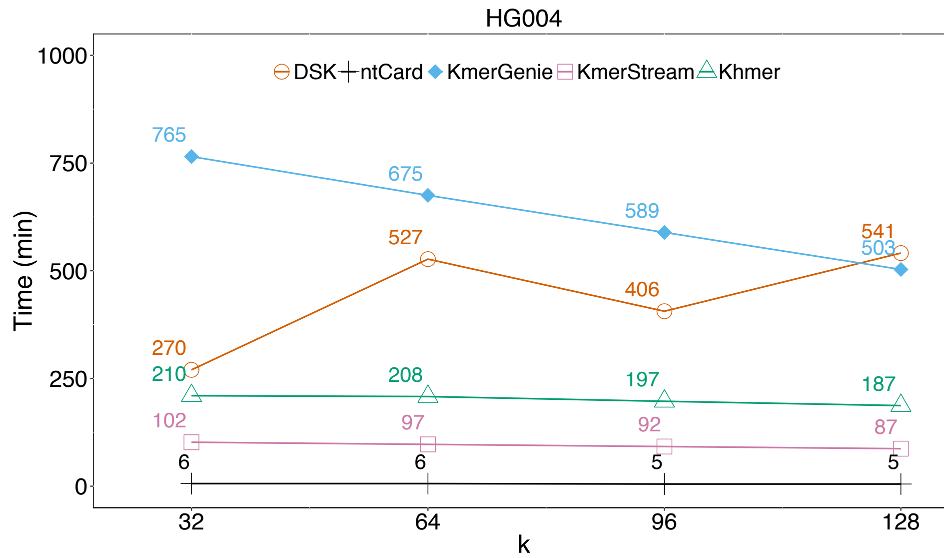

Figure 4.6: Runtime of DSK, ntCard, KmerGenie, KmerStream, and Khmer for HG004 dataset.

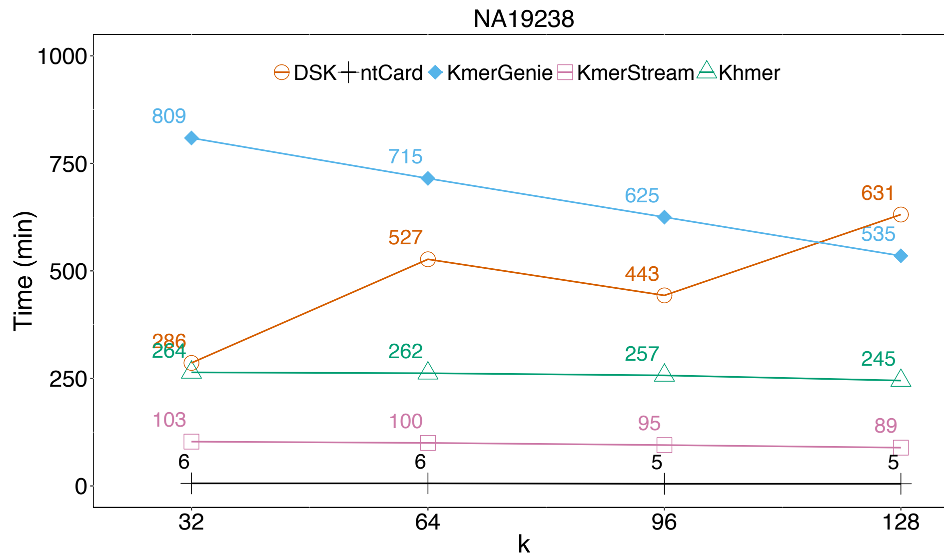

Figure 4.7: Runtime of DSK, ntCard, KmerGenie, KmerStream, and Khmer for NA19238 dataset.

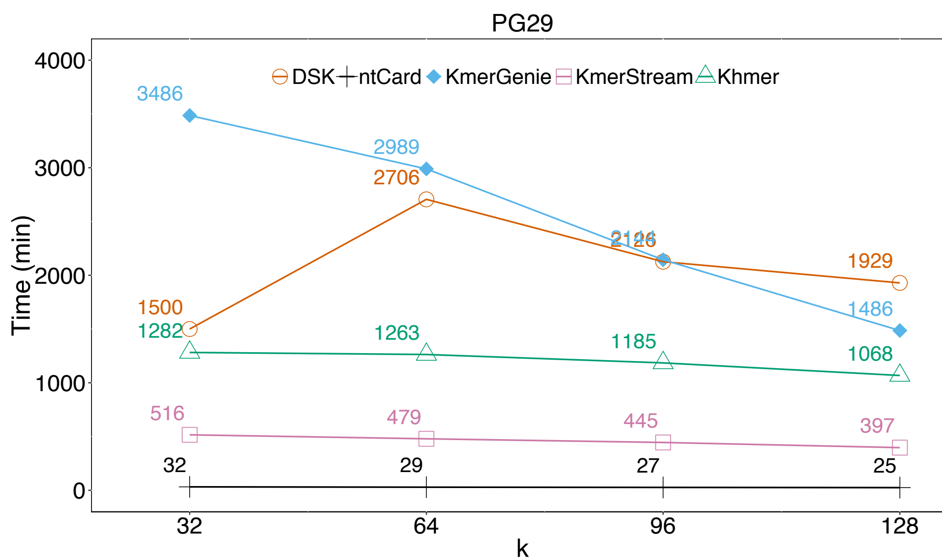

Figure 4.8: Runtime of DSK, ntCard, KmerGenie, KmerStream, and Khmer for PG29 dataset.

only computes F_0 and f_1. In our experiments and computing environment, approximately one third of the ntCard runtime is spent on reading input datasets, and the rest on computing k-mer coverage histograms. Therefore I/O efficiency, which is system and architecture dependent, has a considerable impact on the runtime performance of ntCard.

4.6 Discussion

With growing throughput and dropping cost of the next generation sequencing technologies, there is a continued need to develop faster and more effective bioinformatics tools to process and analyze data associated with them. Developing algorithms and tools that analyze these huge amounts of data *on the fly*, preferably without storing intermediate files, would have many

benefits in a broad spectrum of genomics projects such as *de novo* genome and transcriptome assembly, sequence alignment, repeat detection, error correction, and downstream analysis.

In this work, we introduced the ntCard streaming algorithm for estimating the k-mer coverage frequency histogram for high-throughput sequencing genomics data. It employs the *ntHash* algorithm for hashing all k-mers in DNA/RNA sequences efficiently, samples the k-mers in datasets based on the k-mer hashes, and reconstructs the k-mer frequencies using a statistical model. Using an amount of memory comparable to similar tools, ntCard estimates k-mer frequency histogram for massive genomics datasets, several folds faster than the state-of-the-art approaches.

Sample use cases of ntCard include tuning runtime parameters in de Bruijn graph assembly tasks such as optimal k value for the assembly, and setting parameters in applications utilizing the Bloom filter data structure. ntCard has been used in the new version of our genome assembly software package, ABySS 2.0 [42], to determine the values for total memory size and number of hash functions. It has been also utilized to set the Bloom filter sizes in BioBloom tools [16], which is a general use fast sequence categorization tool utilizing Bloom filters. Using ntCard these tools are able to get the total number of distinct k-mers F_0, as well as the number of k-mers above a certain multiplicity threshold. The k-mer coverage histograms computed by ntCard can be also used as input to utilities like *GenomeScope* (`http://qb.cshl.edu/genomescope/`) for estimating genome sizes, sequencing error rates, repeat contents, and heterozygosity of genomes [13, 71, 75, 96].

We expect ntCard to provide utility in efficiently characterizing certain properties of large read sets, helping quality control pipelines and *de novo* sequencing projects.

Chapter 5

Conclusion

High-throughput sequencing technologies have profoundly altered the scale and scope of research in health and life sciences. As sequencing throughputs continue growing and costs keep dropping, there will be continued needs for accurate and cost-effective algorithms and software tools for the analysis of increasing DNA sequencing data. In health and life sciences research organizations and clinical genomics laboratories, *de novo* assembly and sequence alignment are becoming two key steps in everyday research and analysis. Hence, designing scalable, accurate, and fast algorithms to improve the runtime, memory, and other computational resources in *de novo* assembly pipelines would have a great impact in the field. During my PhD research work, I have designed, developed, and optimized efficient, scalable, and cost-effective algorithms and software tools for the analysis of high-throughput sequencing data specially for sequence alignment and *de novo* assembly problems using state-of-the-art parallel and distributed computing paradigms on high-performance computing infrastructures. Depending on the level of communication required in problems, I utilized different parallel computing paradigms such as "embarrassingly parallel", "loosely coupled, or " tightly coupled" to design efficient and scalable methods to tackle them.

I first introduced DIDA, a distributed and parallel indexing and alignment framework for large-scale sequence alignment tasks. Although there were some solutions for this problem, they still had room for improvement in their runtime and memory usage. Moreover, methods presented in widely popular alignment tools assumed that the target sequence is static and usually represented as a reference genome. The first stage to perform read alignment within those tools involved "indexing" of this reference sequence for faster alignment; a costly stage which is usually discounted in performance measurements. However, there are many applications, where the reference is not static and/or the performance cost of indexing is not negligible. Such cases include resequencing work done on non-model species, and intermediate stages of a *de novo* assembly process. Using DIDA, we were able to address the above challenges and perform large-scale alignment tasks by distributing them across several compute nodes, solving each subtask on a node separately, and finally gathering the partial results into the final output. DIDA was employed in the *de novo* assembly project of the white spruce genome at the Genome Sciences Centre in BC Cancer Agency and enabled us to perform several rounds of large-scale alignment jobs to finish the assembly process.

Later in ntHash, I designed and developed a fast hash algorithm for bioinformatics applications. Many applications in bioinformatics rely on cataloguing or counting DNA/RNA sequences for indexing, querying, and similarity search. These include sequence alignment, genome and transcriptome assembly, RNA-seq expression quantification, and error correction. An efficient way of performing such operations is through the use of hash-

based data structures, such as hash tables or Bloom filters. Thus, improving the performance of hashing algorithms would have a broad impact for a wide range of bioinformatics tools. ntHash achieved this goal by computing hash values for consecutive k-mers in a given sequence using a recursive approach. It was an implementation of cyclic polynomial rolling hashing, and was adapted to the reduced alphabet of DNA sequences. It also efficiently handled computations for reverse-complemented and consequently canonical hash values. Further, ntHash provided a fast way for calculating multiple hash values for a k-mer without repeating the whole hashing procedure for each value - a very useful functionality for bioinformatics applications that utilize the Bloom filter data structure. Our experiments demonstrated significant speed improvements over traditional methods, while maintaining near-ideal distributions for generated hash values. ntHash was employed in a series of software tools to improve the performance of hashing operations such as ABySS, BioBloomTools, ChopStitch, and ntCard.

Finally, I developed a streaming algorithm, called ntCard, for estimating the frequencies of k-mers in genomics data. Many bioinformatics algorithms are designed for the analysis of sequences of some uniform length, conventionally referred to as k-mers. These include sequence alignment tools, and de Bruijn graph assembly software. An efficient algorithm to enumerate the number of unique k-mers, or even better, to build a histogram of k-mer frequencies would be desirable for these tools and their downstream analysis pipelines. Among other applications, estimated frequencies can be used to predict genome sizes, measure sequencing error rates, and tune runtime parameters for analysis tools. However, calculating a k-mer histogram from

large volumes of sequencing data is a challenging task and hence I designed ntCard to tackle these challenges. At its core, ntCard utilized the ntHash algorithm to efficiently compute hash values for steamed sequences. It sampled the calculated hash values to build a reduced representation multiplicity table describing the sample distribution. Finally, it derived a statistical model to reconstruct the population distribution from the sample distribution. The experimental results demonstrated that the ntCard algorithm estimated cardinalities 15× faster, using less memory than the state-of-the-art algorithms, with higher accuracy rates. This makes it as a potentially enabling technology for large-scale genomics applications. ntCard was employed in the new version of our genome assembly software package ABySS 2.0, BioBloomTools, KmerGenie, and ChopStitch algorithms [46].

High-throughput sequencing technologies have extended the frontiers of genomics and bioinformatics research, opening up new doors to investigation and analysis and offering better understanding of broad areas of biology and medicine. Rapid developments and recent advances in DNA sequencing technologies are lowering costs and increasing speeds. With the emergence of third generation sequencing platforms, this continuously developing technology is now being applied within clinical environments, where it has the potential to change treatment and diagnosis outcomes of diseases such as cancer. On the other hand, the third-generation sequencing technology will change the algorithmic landscape. As the read length of the third generation sequencing platforms grows, the related sequencing error increases as well. Therefore, their is a continued need for resource-efficient and accurate algorithms that can handle higher errors and other complex events. It is essential

to tune and adapt existing tools with the need of new sequencing technologies. Inventing new methods that can handle third generation sequencing data efficiently will reduce the computational requirements significantly and consequently speed up and improve the diagnosis and treatment outcomes of diseases.

Bibliography

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. *The Enhanced Suffix Array and Its Applications to Genome Analysis*, pages 449–463. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[2] Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. *Journal of Computer and System Sciences*, 58(1):137 – 147, 1999.

[3] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990.

[4] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting Distinct Elements in a Data Stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*, RANDOM '02, pages 1–10, London, UK, UK, 2002. Springer-Verlag.

[5] David R. Bentley, Shankar Balasubramanian, Harold P. Swerdlow, Geoffrey P. Smith, John Milton, Clive G. Brown, Kevin P. Hall, Dirk J.

Evers, and et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, 11 2008.

[6] Inanc Birol, Justin Chu, Hamid Mohamadi, Shaun D. Jackman, Karthika Raghavan, Benjamin P. Vandervalk, Anthony Raymond, and Ren L. Warren. Spaced Seed Data Structures for De Novo Assembly. *International Journal of Genomics*, 2015.

[7] Inanc Birol, Hamid Mohamadi, Anthony Raymond, Karthika Raghavan, Justin Chu, Benjamin P. Vandervalk, Shaun D. Jackman, and Rene L. Warren. Spaced seed data structures. In *2014 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 15–22, Nov 2014.

[8] Inanc Birol, Anthony Raymond, Shaun D. Jackman, Stephen Pleasance, Robin Coope, Greg A. Taylor, Macaire Man Saint Yuen, Christopher I. Keeling, Dana Brand, Benjamin P. Vandervalk, Heather Kirk, Pawan Pandoh, Richard A. Moore, Yongjun Zhao, Andrew J. Mungall, Barry Jaquish, Alvin Yanchuk, Carol Ritland, Brian Boyle, Jean Bousquet, Kermit Ritland, John MacKay, Jörg Bohlmann, and Steven J.M. Jones. Assembling the 20 Gb White Spruce (Picea Glauca) Genome from Whole-genome Shotgun Sequencing Data. *Bioinformatics*, 29(12):1492–1497, June 2013.

[9] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[10] Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Math.*, 1(4):485–509, 2003.

[11] Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A. Shlyakhter, Matthew K. Belmonte, Eric S. Lander, Chad Nusbaum, and David B. Jaffe. ALLPATHS: De novo assembly of whole-genome shotgun microreads. *Genome Research*, 18(5):810–820, 2008.

[12] Yangho Chen, Tade Souaiaia, and Ting Chen. PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds. *Bioinformatics*, 25(19):2514, 2009.

[13] Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, 2014.

[14] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(1):22, 2013.

[15] Justin Chu, Hamid Mohamadi, Rene L. Warren, Chen Yang, and Inanc Birol. Innovations and challenges in detecting long read overlaps: an evaluation of the state-of-the-art. *Bioinformatics*, 33(8):1261, 2017.

[16] Justin Chu, Sara Sadeghi, Anthony Raymond, Shaun D. Jackman, Ka Ming Nip, Richard Mar, Hamid Mohamadi, Yaron S. Butterfield, A. Gordon Robertson, and Inan Birol. BioBloom tools: fast, accurate and memory-efficient host species sequence screening using bloom filters. *Bioinformatics*, 30(23):3402, 2014.

[17] Jonathan D. Cohen. Recursive Hashing Functions for n-Grams. *ACM Trans. Inf. Syst.*, 15(3):291–320, July 1997.

[18] Thomas C. Conway and Andrew J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.

[19] Graham Cormode and Minos Garofalakis. Sketching Streams Through the Net: Distributed Approximate Query Tracking. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 13–24. VLDB Endowment, 2005.

[20] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58 – 75, 2005.

[21] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[22] Robert C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, 2004.

[23] Lavinia Egidi and Giovanni Manzini. Better spaced seeds using Quadratic Residues. *Journal of Computer and System Sciences*, 79(7):1144 – 1155, 2013.

[24] Brent Ewing, LaDeana Hillier, Michael C. Wendl, and Phil Green.

Base-Calling of Automated Sequencer Traces UsingPhred.?I. Accuracy?Assessment. *Genome Research*, 8(3):175–185, 1998.

[25] Michael Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156, 2006.

[26] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society.

[27] Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight Data Indexing and Compression in External Memory. *Algorithmica*, 63(3):707–730, 2012.

[28] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182 – 209, 1985.

[29] Mahmoud Ghandi, Dongwon Lee, Morteza Mohammad-Noori, and Michael A. Beer. Enhanced Regulatory Sequence Prediction Using Gapped k-mer Features. *PLOS Computational Biology*, 10(7):1–15, 07 2014.

[30] Mahmoud Ghandi, Morteza Mohammad-Noori, and Michael A. Beer. Robust k-mer frequency estimation using gapped k-mers. *Journal of Mathematical Biology*, 69(2):469–500, 2014.

[31] Sante Gnerre, Iain MacCallum, Dariusz Przybylski, Filipe J. Ribeiro, Joshua N. Burton, Bruce J. Walker, Ted Sharpe, Giles Hall, Terrance P. Shea, Sean Sykes, Aaron M. Berlin, Daniel Aird, Maura Costello, Riza Daza, Louise Williams, Robert Nicol, Andreas Gnirke, Chad Nusbaum, Eric S. Lander, and David B. Jaffe. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518, 2011.

[32] Gaston H. Gonnet and Ricardo A. Baeza-Yates. An analysis of the Karp-Rabin string matching algorithm. *Information Processing Letters*, 34(5):271 – 274, 1990.

[33] Sara Goodwin, John D. McPherson, and W. Richard McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nat Rev Genet*, 17(6):333–351, 06 2016.

[34] Faraz Hach, Fereydoun Hormozdiari, Can Alkan, Farhad Hormozdiari, Inanc Birol, Evan E Eichler, and S Cenk Sahinalp. mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nat Meth*, 7(8):576–577, 08 2010.

[35] Yun Heo, Xiao-Long Wu, Deming Chen, Jian Ma, and Wen-Mei Hwu. BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, 30(10):1354, 2014.

[36] David Hernandez, Patrice Franois, Laurent Farinelli, Magne sters, and Jacques Schrenzel. De novo bacterial genome sequencing: Millions of

very short reads assembled on a desktop computer. *Genome Research*, 2008.

[37] Steve Hoffmann, Christian Otto, Stefan Kurtz, Cynthia M. Sharma, Philipp Khaitovich, Jrg Vogel, Peter F. Stadler, and Jrg Hackermller. Fast Mapping of Short Sequences with Mismatches, Insertions and Deletions Using Index Structures. *PLOS Computational Biology*, 5(9):1–10, 09 2009.

[38] Nils Homer, Barry Merriman, and Stanley F Nelson. BFAST: An Alignment Tool for Large Scale Genome Resequencing. *PLOS ONE*, 4(11):1–12, 11 2009.

[39] Lucian Ilie, Hamid Mohamadi, Geoffrey Brian Golding, and William F Smyth. BOND: Basic OligoNucleotide Design. *BMC Bioinformatics*, 14(1):69, 2013.

[40] Piotr Indyk and David Woodruff. Optimal Approximations of the Frequency Moments of Data Streams. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 202–208, New York, NY, USA, 2005. ACM.

[41] Luiz Carlos Irber Junior and C. Titus Brown. Efficient cardinality estimation for k-mers in large DNA sequencing data sets. *bioRxiv*, 2016.

[42] Shaun D. Jackman, Benjamin P. Vandervalk, Hamid Mohamadi, Justin Chu, Sarah Yeo, S. Austin Hammond, Golnaz Jahesh, Hamza

Khan, Lauren Coombe, Rene L. Warren, and Inanc Birol. Abyss 2.0: resource-efficient assembly of large genomes using a bloom filter. *Genome Research*, 27(5):768–777, 2017.

[43] Shaun D. Jackman, Ren L. Warren, Ewan A. Gibb, Benjamin P. Vandervalk, Hamid Mohamadi, Justin Chu, Anthony Raymond, Stephen Pleasance, Robin Coope, Mark R. Wildung, Carol E. Ritland, Jean Bousquet, Steven J. M. Jones, Joerg Bohlmann, and Inan Birol. Organellar Genomes of White Spruce (Picea glauca): Assembly and Annotation. *Genome Biology and Evolution*, 8(1):29–41, 2016.

[44] David S Johnson and Michael R Garey. A 7160 theorem for bin packing. *Journal of Complexity*, 1(1):65 – 106, 1985.

[45] Richard M. Karp and Michael O. Rabin. Efficient Randomized Pattern-matching Algorithms. *IBM J. Res. Dev.*, 31(2):249–260, March 1987.

[46] Hamza Khan, Hamid Mohamadi, Benjamin P. Vandervalk, Ren L Warren, and Inanc Birol. ChopStitch: exon annotation and splice graph construction using transcriptome assembly and whole genome sequencing data. *In revision, Bioinformatics*, 2017.

[47] Stefan Kurtz, Adam Phillippy, Arthur L. Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12, 2004.

[48] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791, 2008.

[49] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nat Meth*, 9(4):357–359, 04 2012.

[50] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[51] Daniel Lemire and Owen Kaser. Recursive n-gram hashing is pairwise independent, at best. *Computer Speech and Language*, 24(4):698 – 710, 2010.

[52] H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *ArXiv e-prints*, March 2013.

[53] Heng Li. BFC: correcting Illumina sequencing errors. *Bioinformatics*, 31(17):2885, 2015.

[54] Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows Wheeler transform. *Bioinformatics*, 26(5):589, 2010.

[55] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078, 2009.

[56] Heng Li, Jue Ruan, and Richard Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11):1851–1858, 2008.

[57] Ming Li, Bin Ma, Derek Kisman, and John Tromp. PatternHunter II: Highly sensitive and fast homology search. *Journal of Bioinformatics and Computational Biology*, 02(03):417–439, 2004.

[58] Ruiqiang Li, Wei Fan, Geng Tian, Hongmei Zhu, Lin He, Jing Cai, Quanfei Huang, Qingle Cai, Bo Li, Yinqi Bai, Zhihe Zhang, Yaping Zhang, Wen Wang, Jun Li, Fuwen Wei, Heng Li, Jun Wang, and et al. The sequence and de novo assembly of the giant panda genome. *Nature*, 463(7279):311–317, 01 2010.

[59] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713, 2008.

[60] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966, 2009.

[61] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, Songgang Li, Huanming Yang, Jian Wang, and Jun Wang. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 2009.

[62] Chi-Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu-Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, Ruiqiang Li, and Tak-Wah Lam. SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878, 2012.

[63] Yongchao Liu, Bernt Popp, and Bertil Schmidt. CUSHAW3: Sensitive and Accurate Base-Space and Color-Space Short-Read Alignment with Hybrid Seeding. *PLOS ONE*, 9(1):1–9, 01 2014.

[64] Yongchao Liu and Bertil Schmidt. Long read alignment based on maximal exact match seeds. *Bioinformatics*, 28(18):i318–i324, 09 2012.

[65] Bin Ma, John Tromp, and Ming Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440, 2002.

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

[67] Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-line String Searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[68] Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[69] Santiago Marco-Sola, Michael Sammeth, Roderic Guigo, and Paolo Ribeca. The GEM mapper: fast, accurate and versatile alignment by filtration. *Nat Meth*, 9(12):1185–1188, 12 2012.

[70] Marcel Margulies, Michael Egholm, William E. Altman, Said Attiya, Joel S. Bader, Lisa A. Bemben, Jan Berka, Michael S. Braverman, Yi-Ju Chen, Zhoutao Chen, Scott B. Dewell, Lei Du, and et al. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, 09 2005.

[71] Guillaume Marais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[72] Paul Medvedev, Eric Scott, Boyko Kakaradov, and Pavel Pevzner. Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, 27(13):i137–i141, 2011.

[73] Colin Meek, Jignesh M. Patel, and Shruti Kasetty. OASIS: An Online and Accurate Technique for Local-alignment Searches on Biological Sequences. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 910–921. VLDB Endowment, 2003.

[74] Páll Melsted and Jonathan K. Pritchard. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, 12(1):333, 2011.

[75] Pll Melsted and Bjarni V. Halldrsson. KmerStream: streaming algorithms for k-mer abundance estimation. *Bioinformatics*, 30(24):3541–3547, 2014.

[76] Hamid Mohamadi. Oligonucleotide Probe Design for Large Genomes using Multiple Spaced Seeds. Master's thesis, McMaster University, 2012.

[77] Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, and Inanc Birol. ntHash: recursive nucleotide hashing. *Bioinformatics*, 32(22):3492–3494, 2016.

[78] Hamid Mohamadi, Hamza Khan, and Inanc Birol. ntCard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, 33(9):1324, 2017.

[79] Hamid Mohamadi, Benjamin P Vandervalk, Anthony Raymond, Shaun D Jackman, Justin Chu, Clay P Breshears, and Inanc Birol. DIDA: Distributed Indexing Dispatched Alignment. *PLOS ONE*, 10(4):1–14, 04 2015.

[80] Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl2):ii79, 2005.

[81] Niranjan Nagarajan and Mihai Pop. Sequence assembly demystified. *Nat Rev Genet*, 14(3):157–167, 03 2013.

[82] Maria Nattestad and Michael C. Schatz. Assemblytics: a web analytics tool for the detection of variants from an assembly. *Bioinformatics*, 32(19):3021–3023, 2016.

[83] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[84] Laurent Noé and Donald E. K. Martin. A Coverage Criterion for Spaced Seeds and Its Applications to Support Vector Machine String Kernels and k-Mer Distances. *Journal of Computational Biology*, 21(12):947–963, 2016/11/23 2014.

[85] Rob Patro, Stephen M Mount, and Carl Kingsford. Sailfish enables alignment-free isoform quantification from RNA-seq reads using lightweight algorithms. *Nat Biotech*, 32(5):462–464, 05 2014.

[86] Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[87] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A Taxonomy of Suffix Array Construction Algorithms. *ACM Comput. Surv.*, 39(2), July 2007.

[88] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer

counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

[89] Rajat Shuvro Roy, Debashish Bhattacharya, and Alexander Schliep. Turtle: Identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, 30(14):1950, 2014.

[90] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *Algorithms for Molecular Biology*, 9(1):2, 2014.

[91] Steven L. Salzberg, Adam M. Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J. Treangen, Michael C. Schatz, Arthur L. Delcher, Michael Roberts, Guillaume Marais, Mihai Pop, and James A. Yorke. GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*, 2011.

[92] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977.

[93] Michael C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363, 2009.

[94] Ariya Shajii, Deniz Yorukoglu, Yun William Yu, and Bonnie Berger. Fast genotyping of known SNPs through approximate k-mer matching. *Bioinformatics*, 32(17):i538–i544, 2016.

[95] Jared T. Simpson. *Efficient sequence assembly and variant calling us-*

ing compressed data structures. PhD thesis, University of Cambridge, 2013.

[96] Jared T. Simpson. Exploring genome characteristics and sequence quality without a reference. *Bioinformatics*, 30(9):1228–1235, 2014.

[97] Jared T. Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 2011.

[98] Jared T. Simpson, Kim Wong, Shaun D. Jackman, Jacqueline E. Schein, Steven J.M. Jones, and Inanc Birol. ABySS: A parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

[99] Andrew D. Smith, Wen-Yu Chung, Emily Hodges, Jude Kendall, Greg Hannon, James Hicks, Zhenyu Xuan, and Michael Q. Zhang. Updates to the RMAP short-read mapping software. *Bioinformatics*, 25(21):2841, 2009.

[100] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[101] Li Song, Liliana Florea, and Ben Langmead. Lighter: fast and memory-efficient sequencing error correction without counting. *Genome Biology*, 15(11):509, 2014.

[102] Henrik Stranneheim, Max Kller, Tobias Allander, Bjrn Andersson,

Lars Arvestad, and Joakim Lundeberg. Classification of DNA sequences using Bloom filters. *Bioinformatics*, 26(13):1595, 2010.

[103] Benjamin P. Vandervalk, Shaun D. Jackman, Anthony Raymond, Hamid Mohamadi, Chen Yang, Dean A. Attali, Justin Chu, Rene L. Warren, and Inanc Birol. Konnector: Connecting paired-end reads using a bloom filter de Bruijn graph. In *2014 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 51–58, Nov 2014.

[104] Benjamin P. Vandervalk, Chen Yang, Zhuyi Xue, Karthika Raghavan, Justin Chu, Hamid Mohamadi, Shaun D. Jackman, Readman Chiu, René L. Warren, and Inanç Birol. Konnector v2.0: pseudo-long reads from paired-end sequencing data. *BMC Medical Genomics*, 8(3):S1, 2015.

[105] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[106] Ren L. Warren, Christopher I. Keeling, Macaire Man Saint Yuen, Anthony Raymond, Greg A. Taylor, Benjamin P. Vandervalk, Hamid Mohamadi, Daniel Paulino, and et al. Improved white spruce (Picea glauca) genome assemblies and annotation of large gene families of conifer terpenoid and phenolic defense metabolism. *The Plant Journal*, 83(2):189–212, 2015.

[107] David A. Wheeler, Maithreyan Srinivasan, Michael Egholm, Yufeng Shen, Lei Chen, Amy McGuire, Wen He, Yi-Ju Chen, Vinod Makhi-

jani, G. Thomas Roth, Xavier Gomes, Karrie Tartaro, Faheem Niazi, Cynthia L. Turcotte, Gerard P. Irzyk, James R. Lupski, Craig Chinault, Xing-zhi Song, Yue Liu, Ye Yuan, Lynne Nazareth, Xiang Qin, Donna M. Muzny, Marcel Margulies, George M. Weinstock, Richard A. Gibbs, and Jonathan M. Rothberg. The complete genome of an individual by massively parallel DNA sequencing. *Nature*, 452(7189):872–876, 04 2008.

[108] Thomas D. Wu and Serban Nacu. Fast and SNP-tolerant detection of complex variants and splicing in short reads. *Bioinformatics*, 26(7):873, 2010.

[109] Daniel R. Zerbino and Ewan Birney. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(5):821–829, 2008.

[110] Justin M. Zook, David Catoe, Jennifer McDaniel, Lindsay Vang, Noah Spies, Arend Sidow, Ziming Weng, Yuling Liu, Christopher E. Mason, Noah Alexander, and et al. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Sci. Data*, 3:160025, Jun 2016.