# BUCKETMAP: SUB-LINEAR HIERARCHICAL DNA READ MAPPING ALGORITHM

by

**GU ZHENHAO**

*(B.Sc., Electrical & Computer Engineering, Shanghai Jiao Tong University)*

**A THESIS SUBMITTED FOR THE DEGREE OF**

**MASTER OF COMPUTING**

in

**COMPUTER SCIENCE SPECIALISATION**

in the

**DEPARTMENT OF COMPUTER SCIENCE**

of the

**NATIONAL UNIVERSITY OF SINGAPORE**

**2023**

Thesis Advisor:
Professor WONG Limsoon

Examiners:
Professor Niranjan NAGARAJAN
Professor ZHANG Louxin

# Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

GU Zhenhao

15 January 2023

*To my dear grandpa*

## Acknowledgments

I would like to thank my thesis advisor, Dr. Wong Limsoon, for his insightful and professional advice that points me in the right direction, both for this thesis and my research career in general. I would also like to thank Dr. Ken Sung Wing-Kin for his course *Combinatorial Methods in Bioinformatics*, which inspired the idea of this dissertation. In addition, I am grateful to Dr. Xin Hongyi from Shanghai Jiao Tong University, who led me into the field of Computational Biology and also shared great thoughts to improve this idea.

Last but not least, I would like to express my deepest gratitude to my parents Gu Gongbin and Wu Suxiang, my family, and my friends, without whose support I could never go this far.

# Contents

# Abstract

BucketMap: Sub-linear Hierarchical DNA Read Mapping Algorithm

by

GU Zhenhao

Master of Computing in Computer Science Specialisation

National University of Singapore

We present a novel cache-efficient, bit-parallel, and alignment-free read mapping algorithm, *BucketMap*, which addresses the challenges of mapping short reads to their locations in a long reference genome with high speed and sensitivity. While current methods rely on large index files for mapping and dynamic programming for calculation of alignment scores, *BucketMap* utilizes a hierarchical, divide-and-conquer mapping strategy along with $k$-mer sampling to achieve sub-linear time in index look-up with respect to the length of reads and the reference genome in both mapping and alignment verification. We also propose a *distinguishability filter* and a *quality filter* that help us select high-quality $k$-mers and achieve high sensitivity and selectivity. Our experiment results show that BucketMap requires up to 2-5 times less memory and 2-6 times faster than the state-of-the-art mappers while maintaining high sensitivity and precision.

**Keywords:** DNA Read Mapping, Reference Genome, String Alignment, Next Generation Sequencing, Randomized Algorithm, Sub-linear Algorithm, Cache-oblivious Algorithm.

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

With the fast development of Next Generation Sequencing (NGS), and currently the Third Generation Sequencing technology, the sequencing of DNA has become cheaper and faster than ever. The cost of sequencing a human genome has decreased to as low as 1,000 US dollars [49]. The Illumina platforms, for example, are able to produce 2.4 billion 300-base-pair-long short reads in 48 hours [20].

The high speed and cost of genome sequencing have led to many commercial uses. With the sequencing data, we are able to identify the breed composition and traits of various animals, diagnose genetic disorders and rare diseases, and provide health information to the general public. Current databases of DNA reads also contain a huge number of reads and observe an increase in read length. It is becoming increasingly crucial that the sequencing data is efficiently and accurately analyzed.

The first step of processing sequencing data is DNA read mapping. That is, we want to map the millions of sequenced short reads to their locations in the long reference genome (usually billions of base pairs in length). The problem of how to efficiently and correctly do the mapping has been a fundamental and well-studied topic in the field of Computational Biology.

The read mapping problem has a wide range of applications. Theoretically speaking, the read-mapping problem is related to several other fields in Computer Science, including the classic string-searching algorithm, calculation and approximation of edit distance, and fast database querying. Practically speaking, the results of read-

mapping provide essential information for a variety of downstream analyses, such as RNA-seq, genome assembly, single nucleotide polymorphism (SNP) detection, structural variation detection [15] and somatic variant detection [23], giving critical insight into genetic disorder, cancer mutations, evolutionary relationship, etc. Therefore, it is crucial to achieving high speed and sensitivity in read mapping.

The main challenges of this problem lie in

- **High memory consumption during indexing and mapping**. The reference genome is usually long. For instance, the human reference genome typically contains $3.055 \times 10^9$ base pairs (bp) [34] and takes several gigabytes (Gb) to store. The index files, which map reads to their locations in the reference genome, are inevitably large and consume a large amount of memory. This leads to a lot of cache misses and page faults when querying the locations of reads.

- **Toleration of sequencing errors and mutations**. The main difference between the read mapping problem and the classic string matching problem is that the reads may differ from their origins in the reference genome. It can be several substitutions, insertions, and deletions in the case of sequencing errors and SNPs, or a large region of mismatch in the case of structural variation. The potential errors and mutations should be taken into consideration during the mapping.

- **Needs for high speed and sensitivity during alignment verification**. For effective downstream analysis such as variant calling, we would want the reads that are not coming from repetitive regions to be mapped to the correct unique location in the ideal case. A fast alignment verification tool is needed to eliminate the false positive locations as much as possible.

## 1.2 Thesis Objective

To overcome the challenges stated in section 1.1, we aim to propose a new DNA read mapping algorithm *BucketMap*, that is different from other mappers in that

1. it adopts a novel hierarchical, divide-and-conquer DNA read mapping strategy that we only need to keep one small index file in the memory at a time.

2. it utilizes sampling of $k$-mers from the reads to achieve sub-linear time for index look-ups in mapping and alignment verification.

*BucketMap* has a faster speed than many other mappers due to the following three features.

1. **Cache-efficient**. The index file we keep in the memory only takes about 200 Kb of space, which can easily fit into the cache of an ordinary computer, facilitating fast querying of positions of reads.

2. **Bit-parallel**. We propose a new data structure called *bucket filter* (Algorithm 1) that uses bit-wise operations to facilitate faster elimination of false regions that cannot contain the read.

3. **Alignment-free**. We propose another way ($(\varepsilon, k)$-matching) than the edit distance or alignment score to verify the goodness of alignment, which allows us to skip the dynamic programming step.

## 1.3 Thesis Organization

The rest of this thesis is organized as follows. We provide a review of related and state-of-the-art algorithms and data structures solving the read mapping problem in chapter 2, and a detailed description and analysis of our new mapper in chapter 3. The implementation details as well as the benchmark results are shown in chapter 4. Finally, we conclude the entire thesis and discuss possible directions for future research in chapter 5.

To avoid confusion, we summarize the commonly used mathematical notations in this thesis in Table 1.1.

**Table 1.1:** Mathematical Notations used in the paper.

| Notation | Meaning |
|---|---|
| $\Sigma$ | The set of letters in the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ |
| $n$ | Length of the reference genome |
| $m$ | Maximum length of a bucket |
| $\mathcal{B}$ | The set of all buckets |
| $r$ | Length of the reads |
| $k$ | Length of the seeds (sub-string of reads used for mapping) |
| $s$ | Number of seeds we draw from the query short read |
| $Q$ | Query matrix, where $Q_{i,j}$ stores the occurrence of $k$-mer $j$ in bucket $i$ |
| $P$ | The pattern or read that we want to search for its locations |
| $S$ | Sub-string of the text (reference genome) that can be matched with $P$ |
| $\lvert \cdot \rvert$ | Size of a set, a list, or a string. |

# Chapter 2

# Related Work

As both the number of sequences and read length grow fast in sequence databases, it is crucial that read mapping algorithms achieve high speed, sensitivity, and selectivity. Over the past few decades, numerous studies have already been done to solve the problem of read mapping.

The workflow of current methods can be generally divided into two steps: (1) Indexing and mapping: based on the sub-strings (or *seeds*) in the read, we rapidly filter out regions in the reference genome that cannot contain the read. (2) Pairwise Alignment: among the remaining candidate regions, we perform pairwise string alignment to select the best regions where the read and the reference genome can be best matched together, and identify potential spots of mutations.

## 2.1 Indexing and Mapping

Two kinds of indexing are commonly used to help map seeds to their candidate locations [9]: the *k*-mer index based on hashing, and FM-index [13] based on Burrows-Wheeler transform (BWT) [8] and suffix arrays [33]. Both data structures serve to find the location/s of seeds in the read.

The hashing-based methods usually use a hash table that stores key-value pairs, where keys are seeds of length $k$ (or *k-mers*) and the values are their locations in the read or the reference genome. Some earlier algorithms, such as *MAQ* [30] and *RMAP* [41], build multiple hash tables for the reads, and scan the whole reference genome against the hash tables to find hits. Those hash tables are smaller and take up little memory, but the need to scan through the reference genome multiple

times slows down these algorithms. On the other hand, *MrFast* [2], *MrsFast* [16], *FastHASH* [50], *GRIM-Filter* [21], *Minimap2* [28] and many other recent mappers rely on a hash table capturing the seeds in the entire reference genome. Those hash tables are typically larger than the original reference genome file, using at least $\mathcal{O}(n)$ space, but the average time needed to query a seed is $\mathcal{O}(1)$.

The BWT-based methods consist of some of the most popular DNA read alignment software, including *Bowtie* [26] and *Bowtie2* [25], *BWA-MEM* [29] and *BWA-MEM2* [46], etc. They work based on FM-index, a suffix-array-like data structure that captures the BWT-compressed reference genome, allowing fast counting of occurrences of patterns and exact string matching. The FM-index is typically taking sub-linear space that is proportional to the compressed text. For instance, it takes only 1.3 Gb of memory for the human genome [26]. Moreover, the time needed to find the location of a seed of length $r$ is only $\mathcal{O}(r + \texttt{occ} \log^{1+\varepsilon} n)$ [12], where $\texttt{occ}$ is the number of times the seed occurs in the reference genome and $\varepsilon < 0.01$ is an implementation-related constant.

Generally, a read mapper breaks each read into one or several seeds and tries to find their possible locations in the reference genome using the above index files. There are two strategies that utilize those candidate locations to finally determine the exact location of the entire read.

1. The *seed-and-extend* approach, in which case the locations of the read are determined by the locations of one longer seed. The mapper takes the candidate locations of one seed, extracts the corresponding DNA sequence in the reference genome, and then compares the sequence with the read using approximate string matching algorithms, which will be discussed in the next section.

   This is a popular approach for many mappers such as FastHASH, Bowtie2, and BWA, as both the $k$-mer index and FM-index are fast at finding exact matches. This approach is fast as it only requires a few queries to the index file, but it may fail to map a read if all of the seeds are contaminated with sequencing errors.

2. The *seed-and-vote* approach, in which case the locations of the read are jointly

determined by the locations of multiple seeds. This approach is used extensively in many mappers such as the $q$-gram filter [37] and RazerS 3 [48], which find candidate regions that share a sufficient number of seeds using the $q$-gram lemma and a sliding window technique [9].

Liao et. al. proposed an alternative approach where each seed will vote for possible locations of the read, and the locations that get the most votes are identified as the final mapping location.[32], is used extensively in the *Subread* package. It is very fast and error-tolerant.

Current mappers are already quite fast and accurate by using the above approaches. However, they still have room for improvement. In particular, all of the mappers are using a few large index files that don't typically fit in the cache of an ordinary computer. As a result, the querying for seeds, which requires multiple jumps inside the index, may lead to many cache misses or page faults, slowing down the mapping process.

## 2.2 Pairwise Alignment

The question of how to evaluate the alignment between the read and the substrings of the reference genome is also a very well-studied topic. Traditionally, the similarity of two strings is measured by the *Hamming distance* [39], which simply counts the number of mismatches between the two strings, and the *Levenshtein distance* (or *edit distance*) [27], which finds the minimum number of edits (substitutions, insertions or deletions) needed to transform the reference string $S$ to the read string $P$.

On top of the edit distance, several more sophisticated scoring schemes have been proposed to quantify the similarity that better captures the nature of mutations. For example, the BLAST matrix [3] and the transition-transversion matrix that gives different penalties for mismatches and matches, and the gap affine penalty scheme [25] that penalizes insertions and deletions (indels) using an affine function $\alpha + q\beta$, where $\alpha > 0$ is the gap opening penalty, $q$ is the number of consecutive indels, and $\beta$ is the gap extension penalty.

Despite variations in scoring schemes, the optimal alignment between the two strings that minimizes the penalty score can be found using variants of the Smith-Waterman algorithm [42] in $\mathcal{O}(r^2)$ time [14]. If it is known that the number of errors is less than $e$, we can perform a banded version of the Smith-Waterman algorithm or the Landau-Vishkin algorithm [24], reducing the alignment time to $\mathcal{O}(e^2 r)$.

Mappers using the seed-and-extend approach use different kinds of scoring schemes to find the optimal alignment and judge the goodness of mapping based on the alignment score. This process is slow and a bit unnecessary because

1. The pairwise alignment takes time that is in practice (as $e$ is kept small in the context of read mapping) linearly dependent on the length of the read, which is slow if we are to apply these algorithms to the Third generation sequencing databases.

2. The optimal alignment generated by dynamic programming algorithms might not actually reflect the truth of how the sequences have mutated.

3. Downstream analyses, such as variant calling, sometimes perform re-alignment to further eliminate false-positive alignments [22].

Therefore, in this project, we aim to address the above two problems by proposing a new hashing-based mapping algorithm that

- minimizes the index file size, and

- uses a variant of the seed-and-vote strategy to avoid pairwise alignment, while still keeping a high selectivity.

# Chapter 3

# BucketMap

## 3.1 Problem Formulation

The read mapping problem, which is basically a variant of the classic string matching problem while taking errors into consideration, is usually written in the following form [9],

**Definition 1** ($e$-errors problem)**.** Given text $T$ and pattern $P$, the *$e$-errors problem* asks for all sub-strings of $T$ that can be converted to $P$ using up to $e$ substitutions, insertions, or deletions, i.e. have an edit distance to $P$ within $e$.

This definition of the problem is unfortunately a bit outdated. The latest sequence alignment algorithms are using much more complicated scoring metrics than the edit distance, such as the transition-transversion matrix as the penalty for substitutions and the affine gap penalty for insertions and deletions. The scoring metric can vary a lot among different applications and tools, and the simple representation of the $e$-errors problem in terms of edit distance is not sufficient.

We hereby propose a more unified and biologically-meaningful way of quantifying the goodness of alignment, not by counting how many errors there are between the two strings, but by measuring how well the parts of two strings can be matched together.

**Definition 2** ($k$-matching)**.** Given text $S$ and pattern $P$, The *$k$-matching* $\mathcal{M}_k(S, P)$ is the longest common sub-sequence between the ordered sequences of $k$-mers in $S$ and in $P$.

An example of 3-matching between two strings $S = \texttt{ATCGGACTTTAGA}$ and pattern $P = \texttt{ATCGGTTTAGA}$ is shown in Figure 3.1.

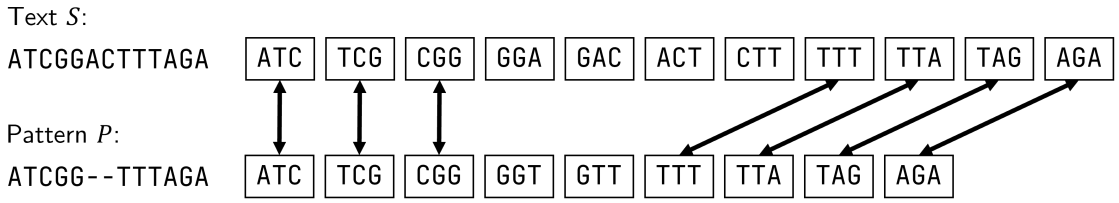


**Figure 3.1:** An artifical example of $k$-matching between text $S = \texttt{ATCGGACTTTAGA}$ and pattern $P = \texttt{ATCGGTTTAGA}$, with $k = 3$. The ordered sequences of $k$-mers in $S$ and in $P$, along with their translated amino acids, are listed in the boxes. Due to the two deletions, only 7/11 of all $k$-mers in $S$ can be matched (as annotated using the arrows) with the ones in $P$. The maximum matching is therefore $\mathcal{M}_k(S, P) = (\texttt{ATC}, \texttt{TCG}, \texttt{CGG}, \texttt{TTT}, \texttt{TTA}, \texttt{TAG}, \texttt{AGA})$.

Intuitively speaking, the size of $k$-matching captures the parts of $P$ where it matches the parts in $S$. $|\mathcal{M}_k(S, P)|$ being closer to the number of $k$-mers in $P$ means a better alignment between $P$ and $S$. This definition is similar to *LCSk* [5], which is defined to be the maximal number of $k$ length sub-strings matching in both strings. Our definition differs from LCSk in that $k$-matching allows overlapping between two adjacent $k$-mers in the matching.

This new definition of the goodness of alignment has the following properties:

- A larger $k$ implies a higher penalty for the errors. Each error (mismatch or indel) will affect up to $k$ of all the $k$-mers, excluding them from the $k$-matching of the two strings.

- Errors that are close to each other get penalized less than the case if they are far apart. For the example in Figure 3.1, we have two consecutive deletions and only 4 $k$-mers are excluded from the $k$-matching. If those two deletions are far from each other, they can affect up to 6 $k$-mers. In this sense, the $k$-matching scheme is quite similar to the gap affine penalty.

  $k$-matching still differs from the gap affine penalty in that

  1. $k$-matching punishes less for gaps (insertions and deletions) that appear within a range of length $k$, while the gap affine penalty punishes less only for consecutive insertions or deletions.

2. $k$-matching punishes less for sequencing error in the length of *homopolymers*, consecutive repetitions of the same base. This type of error is very common in Nanopore sequencing [11]. An example is shown in Figure 3.2.

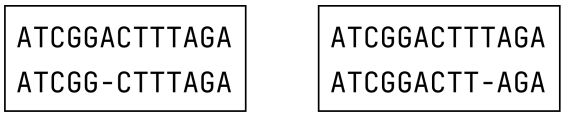


**Figure 3.2:** Example of $k$-matching ($k = 3$ in this case) punishing less for errors in the length of homopolymers. **Alignment on the left**: A regular deletion of a base `A`, excluding three 3-mers `GGA`, `GAC` and `ACT` from the $k$-matching and resulting in a penalty of $-3$. **Alignment on the right**: A deletion among a homopolymer, excluding only one 3-mer `TTT` and resulting in a penalty of only $-1$.

With this new definition, we can propose an alternative definition of the read mapping problem.

**Definition 3** ($(\varepsilon, k)$-matching problem)**.** Given text $T$ and pattern $P$, and given a positive integer $k \leq \min\{|S|, |P|\}, \varepsilon \in (0, 1]$, the $(\varepsilon, k)$-*matching problem* asks for all shortest sub-strings $S$ of $T$ such that the $k$-matching between $S$ and $P$ takes up a large proportion of all $k$-mers. In particular,

$$\frac{|\mathcal{M}_k(S, P)|}{\max\{|S|, |P|\} - k + 1} \geq \varepsilon.$$

In other words, If $S$ and $P$ are to be matched, then we expect them to share a large number of $k$-mers.

## 3.2 Method

We are now ready to propose our new algorithm, *BucketMap*, which finds solutions to the $(\varepsilon, k)$-matching problem in $\mathcal{O}(\varepsilon^{-4}|\mathcal{B}| \log^2 |\mathcal{B}| \log \delta^{-1} + \varepsilon^{-2} \log m \log \delta^{-1})$ time with a sensitivity and specificity of at least $1 - \delta$. If we fix the bucket size to be large ($m = \Theta(n)$), then the number of buckets would be small ($|\mathcal{B}| = \mathcal{O}(1)$), and the total time will be bounded by $\mathcal{O}(\varepsilon^{-4} \log \delta^{-1} + \varepsilon^{-2} \log n \log \delta^{-1})$, which is sub-linear with respect to the length of both the read and the reference genome.

### 3.2.1 Overview

Our new mapping algorithm *BucketMap*, named after the classic *bucket sort* algorithm, utilizes a hierarchical mapping strategy. The basic idea is to divide the

reference genome into a set $\mathcal{B}$ of overlapping buckets. Then, instead of finding the location/s of each read in the entire reference genome directly, we first find the candidate buckets that might contain each read. Finally, within each bucket $b \in \mathcal{B}$, we find the exact locations of all the reads that are mapped to $b$.

*BucketMap* achieves good performance in both time and memory usage during mapping due to

- the smaller index files used when finding the exact locations of the reads. With the hierarchical mapping strategy, we no longer build a single large index file for the whole genome. Rather, we only need to keep the index for sequences in one specific bucket in the memory at a time, which is usually just taking several hundred kilobytes (Kb) of space and can easily fit in the cache of an ordinary computer, facilitating fast query and alignment verification.

- sampling of $k$-mers from the read. It turns out, in our analysis in section 3.3, that we don't need to read and process the entire DNA read string to correctly do the mapping and verification of alignment – A sub-linear number of $k$-mers drawn from the read is sufficient. Using a sampling strategy, our mapping algorithm achieves sub-linear time with respect to the read length and genome length, and proves efficient, especially for longer reads.

*BucketMap* contains three main components,

1. **Indexer**, which divides the reference genome into a set $\mathcal{B}$ of overlapping buckets (each containing $m$ base pairs), and builds a hash map $M : \Sigma^r \to \mathcal{P}(\mathcal{B})$ that maps a read of length $r$ to a subset of buckets in $\mathcal{B}$.

2. **Mapper**, which uses the map $M$ to find all candidate buckets that might contain each read.

3. **Locator**, where we iterate through all buckets. For each bucket $b_i$, we build a $k$-mer index hash table and use it to map all reads to their exact locations within the bucket, as well as perform alignment verification and mapping quality estimation.

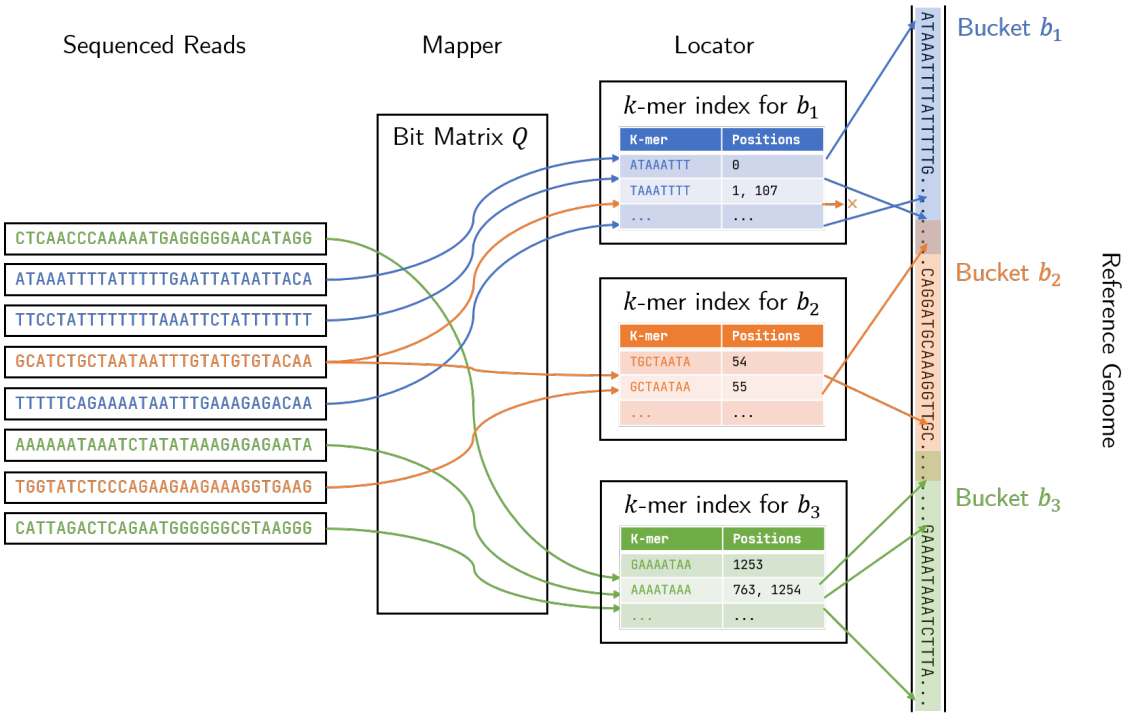


**Figure 3.3:** Demonstration of a basic workflow of BucketMap. The **reference genome** (shown on the right) is divided into several overlapping buckets $\{b_1, b_2, b_3, \dots\}$. The **sequenced reads**, colored with the bucket it actually came from, first goes through the **mapper** which assigns each read to their candidate buckets. Finally, for each smaller bucket, the **locator** maps all reads assigned to it to the exact location within the bucket and discards the invalid assignments.

An illustration of the basic workflow of BucketMap is shown in Figure 3.3. In the following sections, we discuss the details of the algorithms used for the three components.

### 3.2.2 Indexer

The indexer takes the reference genome as input and outputs a bit matrix $Q$ that is later used by the mapper to map each read to their candidate buckets. The job of the indexer involves two steps:

1. **Bucket division**. With a given positive integer $m$ and the maximum length of reads $r$, we can divide the reference genome of length $n$ into $\lceil n/m \rceil$ buckets with the length of each bucket being at most $m + r$. A read $P \in \Sigma^r$ can only be mapped to a bucket $b_i$ if it is contained in $b_i$ **entirely**. An illustration of how the reference genome is divided can be seen in Figure 3.4.

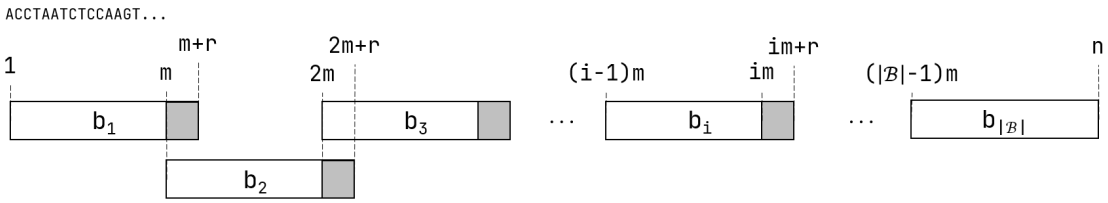


**Figure 3.4:** Illustration of how we divide the reference genome of length $n$ into the set $\mathcal{B}$ of overlapping buckets, $\{b_1, b_2, \ldots, b_{|\mathcal{B}|}\}$. The gray parts indicate regions where bucket $b_i$ is overlapping with the next bucket $b_{i+1}$.

Though developed independently, we noticed that the bucket division approach is similar to the one used by QUASAR [7]. Our way of bucket partition, which limits the length of each overlapping region to be $r$ (instead of $m/2$ in QUASAR), minimizes the chance of a read being mapped to two adjacent buckets simultaneously while ensuring every read under length $r$ can be contained in at least one bucket entirely.

2. **Index building**. To quickly filter out the buckets that don't contain a feasible solution, we use an alignment-free approach and ignore the order information of the $k$-mers. We construct a bit matrix $Q \in \text{Mat}(\{0,1\}, |\mathcal{B}| \times |\Sigma|^k)$, where

$$Q_{i,j} = \begin{cases} 1 & \text{if } k\text{-mer } j \text{ is present in bucket } b_i \\ 0 & \text{otherwise} \end{cases}.$$

Now, suppose a read $P$ has the sequence of $k$-mers $(p_1, p_2, \ldots, p_{|P|-k+1})$. If $\sum_{j=1}^{|P|-k+1} Q_{i,p_j}$ is small, then we can conclude that the bucket $i$ shares few $k$-mers with $P$ and therefore we can eliminate bucket $i$.

Building this bit matrix requires a linear scan of the reference genome, using $\mathcal{O}(n)$ time. The resultant bit matrix $Q$ will take $|\mathcal{B}| \times |\Sigma|^k = \lceil \frac{n}{m} \rceil \times |\Sigma|^k$ bits to store. If we choose the size of each bucket $m = |\Sigma|^{k-1}$, then the space usage would be $\mathcal{O}(n \cdot |\Sigma|)$ bits, which is also linearly dependent on text size.

In subsection 3.2.3, we show that we can use the sampling of $k$-mers and bit-parallel operations instead of summations to speed up the filtration.

### 3.2.3 Mapper

The mapper takes the bit matrix $Q$ and all the reads as input, maps the reads to their candidate buckets, and outputs a vector $R$ of length $|\mathcal{B}|$, where $R_i$ contains the set of all reads that are mapped to bucket $i$. The basic workflow of the mapper is demonstrate in Figure 3.5.

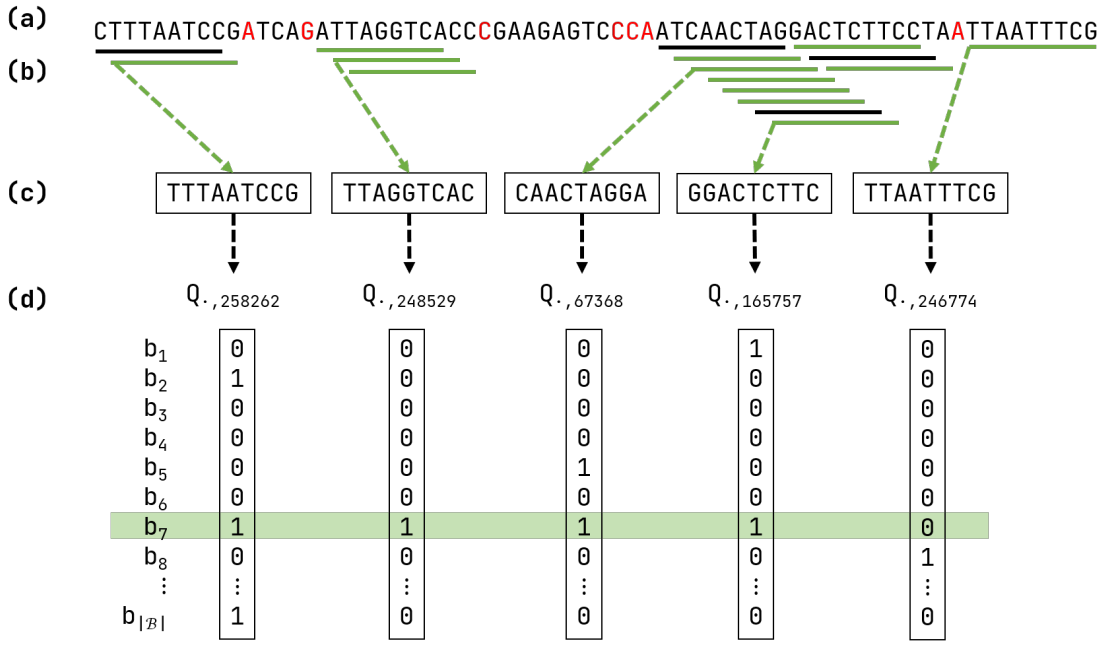


**Figure 3.5:** Workflow of the mapper with $k = 9$. **(a)** The original short read. The bases marked red are the ones with low base quality scores. **(b)** The *quality filter* collects all $k$-mers with high total quality scores (underlined with black or green lines), essentially avoiding the low-quality base calls. The *distinguishability filter* further eliminates the $k$-mers (underlined with black lines) that appear in more than half of the buckets. **(c)** We sample several $k$-mers out of those high-quality and highly-distinguishable $k$-mers. **(d)** Each sampled $k$-mer is converted to an integer $j$, and we look at the $j^{\text{th}}$ column in $Q$. The row/s with the most 1's in all of the sampled columns of $Q$ ($b_7$ in our example) is/are identified as candidate bucket/s.

Its job can also be divided into two steps.

1. ***k*-mer sampling**. For candidate bucket filtering, we may go through $k$-mers from the read and look for their presence in the buckets. However, not all $k$-mers are reliable. We apply two additional $k$-mer filters:

   - A *quality filter*, which sums up the base quality scores (q-score) in each $k$-mer, and eliminates the $k$-mers with low total q-score. This is based on the observation that base calls with low q-score are more likely to

contain sequencing errors [36], since the score is basically calculated by $q = -10 \log p$, where $p$ is the probability of base calling error. The quality filter allows the mapper to sample error-free $k$-mers as much as possible, which allows high sensitivity even with the presence of sequencing errors.

According to our experiments using Illumina reads, the quality filter is able to increase sensitivity by about 10% with the same sample size.

- A *distinguishability filter*, which filters out the $k$-mers that appear in more than half of the buckets, i.e. $k$-mers in the set $\{j : \sum_{i=1}^{|\mathcal{B}|} Q_{i,j} \geq |\mathcal{B}|/2\}$. This ensures that the $k$-mers we sample can help us eliminate at least half of the buckets so that approximately $\mathcal{O}(\log |\mathcal{B}|)$ samples (assuming independence among the samples) from the $k$-mers are sufficient to reduce to $\mathcal{O}(1)$ candidate buckets that contain all the $k$-mer samples. In short, the distinguishability filter help guarantees high selectivity even with a small sample size.

  According to our experiments, the distinguishability filter can reduce the number of candidate buckets returned by the mapper algorithm by half, with the same sample size.

2. **Bucket filtering**. Now that we have the sampled $k$-mers $(j_1, j_2, \ldots, j_s)$ and their corresponding columns in the bit matrix $Q$, our goal is to find buckets that contain the most sampled $k$-mers, i.e. the rows on which there are the most 1's in the sampled columns.

   A trivial answer would be to add the columns $Q_{\cdot,j_1}, Q_{\cdot,j_2}, \ldots, Q_{\cdot,j_s}$ together and find the maximum element. However, the necessity of using an integer (instead of a bit) to store the sum for each bucket and the lower speed of summation may slow down the filtering.

   We hereby propose a new bit-parallel data structure, *bucket filter*, that quickly sorts out buckets that miss fewer than $e$ $k$-mers among the $s$ samples. The basic idea is to use bit vectors $F_0, F_1, \ldots, F_e \in \{0,1\}^{|\mathcal{B}|}$, where the set bits in $F_i$ stands for the buckets that miss exactly $i$ of the $k$-mers.

   As shown in Algorithm 1, the selection of candidate buckets that miss fewer

**Algorithm 1:** Bucket Filter

**Input :** $s$ bit vectors of size $|\mathcal{B}| \times 1$: $Q_{\cdot,j_1}, Q_{\cdot,j_2}, \ldots, Q_{\cdot,j_s}$, error threshold $e$.
**Output :** Indices of rows at which we observe more than $s - e$ 1's in all $s$ bit vectors, i.e. buckets that miss fewer than $e$ of the $k$-mer samples.

```
1 for i ← 0 to e do
2     F_i ← 1_{|B|×1};                    // Bit vectors of length |B| filled with 1's
3 for each j in (j_1, j_2, ..., j_s) do
4     for i ← e down to 1 do
5         F_i ← F_i & (F_{i-1} | Q_{·,j});    // '&' is bitwise-AND, '|' is bitwise-OR
6     F_0 ← F_0 & Q_{·,j};
7 return indices of set bits in F_e;
```

than $e$ $k$-mer samples can be done using bitwise-AND and OR operations, effectively reducing the depth of the parallel algorithm to $\mathcal{O}(s \cdot e)$ while the total work is $\mathcal{O}(|\mathcal{B}| \cdot s \cdot e)$, where $s$ is the sample size and $e$ is the maximum number of misses allowed. The choices of $s$ and $e$ are discussed in section 3.3. As a by-product, we can find the buckets that miss fewer $k$-mers by checking the content of $F_0, F_1, \ldots, F_{e-1}$.

In the actual implementation, we don't return all the set bits in $F_e$ as in Algorithm 1. Instead, we find the first bit-vector in $F_0, F_1, \ldots, F_e$ that is not $\mathbf{0}$, and return the indices of set bits in that vector. This corresponds to the buckets that **contain the most of sampled $k$-mers**. We later prove in Corollary 1 that with high probability, we are going to return only the correct bucket if the read can be mapped uniquely.

Now that we can efficiently map reads to their candidate buckets with the alignment-free method that ignores the relative positions of the $k$-mers. We show in subsection 3.2.4 that how we can use the positional information of $k$-mers to find the exact locations of the read and do alignment verification.

### 3.2.4 Locator

The locator takes the reference genome and the output of the mapper (the vector $R$ with each element $R_i$ containing all reads that are mapped to a bucket $b_i$)

as input, and produces a SAM file output that summarizes the mapping.

The idea of the locator is similar to the *seed-and-vote* scheme [32]. However, our locator avoids generating an alignment when evaluating locations, allowing faster location filtering. For each bucket $b_i$, we go through each read $P \in R_i$ that is mapped to it. Similar to the mapper, we again sample some high-quality $k$-mers (using the quality filter as in subsection 3.2.3).

Each sampled $k$-mer $i$ casts its vote on the possible starting position of the read, which is jointly determined by

- the positions it appears in the reference genome $p_1, p_2, \ldots, p_l$,
- its position in the read $\hat{p}_i$.

Then, the $k$-mer will judge $p_1 - \hat{p}_i, p_2 - \hat{p}_i, \ldots, p_l - \hat{p}_i$ as possible starting positions of the read. It will cast its vote for these positions as well as their neighboring positions to allow a number of insertions or deletions. Each position is going to receive at most one vote from a sampled $k$-mer.

Finally, the position that gets the most number of votes that is above some threshold wins and is identified as the exact location of the read. The basic workflow of the above process is illustrated in Figure 3.6.

Due to the need to go through the reads that are mapped to each bucket, it is the most convenient and fastest to store all reads in the memory. However, storing the original reads might be very memory-consuming. In the actual implementation of BucketMap, we sample $k$-mers from the reads before running the locator algorithm and store only the sampled $k$-mers (mapped to integers) in memory. If we take 10 samples from each read, it takes only 57 Mb of space to store $k$-mers of one million reads, which is independent of read length and quite acceptable even for a larger number of reads.

## 3.3 Algorithm Analysis

To formalize the ideas of mapper and locator, we consider a variant of the property testing of distributions problem [1],

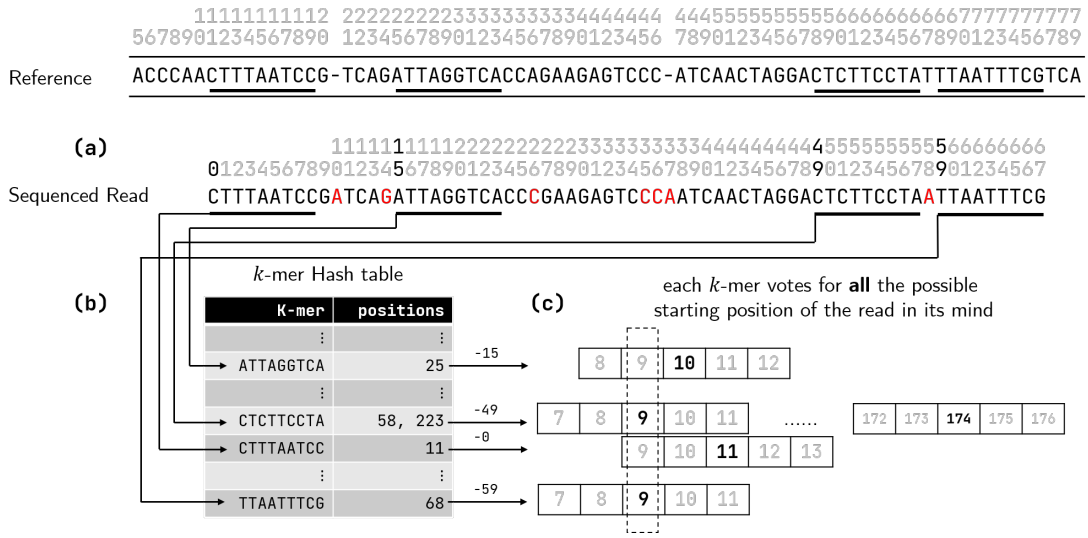


**Figure 3.6:** Illustration of the workflow of the locator. **(a)** The sequenced read containing two insertions and two substitutions, along with its alignment with the reference genome. **(b)** Several $k$-mers are sampled, again avoiding the low-quality bases (marked in red). We query the locations of the $k$-mers using the $k$-mer index hash table of that bucket. **(c)** Each $k$-mer casts its vote for the possible starting positions of the whole read. The $k$-mer `ATTAGGGTCA` appears at position 25 in the reference genome, and at offset 15 in the read. Therefore, it will think the starting position of the read to be around $25 - 15 = 10$. It casts its vote for 10 as well as the neighboring positions to allow a number of indels. Finally, the smallest number that has the most votes (9 in this case) wins and is identified as an approximate position for the read.

**Definition 4** (Treasure Digging Problem)**.** Given sample access to a set $\mathcal{B}$ of independent Bernoulli random variables $X_1, X_2, \ldots, X_{|\mathcal{B}|}$, with one of the random variables $X^* \in \mathcal{B}$ has expectation $\mathsf{E}[X^*] = p + \varepsilon$ and $\forall_{X \in \mathcal{B} \setminus \{X^*\}} \mathsf{E}[X] = p$, where $\varepsilon > 0$, how many samples are needed to draw from each random variable to find $X^*$?

Here, $X^*$ is the "treasure", or the candidate bucket or location we want to find, while the other random variables $X \in \mathcal{B} \setminus \{X^*\}$ represent the other false buckets or locations. We would like to see how far we need to "dig", or how many $k$-mer samples we need to draw from the read so that we can correctly distinguish the "treasure" from other false positions with high probability.

Since $\mathsf{E}[X^*] > \mathsf{E}[X]$ for all $X \in \mathcal{B} \setminus \{X^*\}$, a natural strategy is to get $s$ samples from each random variable and see which one has the most 1's in its samples, as shown in Algorithm 2.

Notice that Algorithm 1 is simply a vectorized and bit-parallel version of this treasure-digging algorithm. We now prove an upper bound for the sample size needed.

**Algorithm 2:** Treasure Digging

**Input :** The set of random variables $\mathcal{B} = \{X_1, \dots, X_{|\mathcal{B}|}\}$, the sample size $s$, and a threshold $t$ to distinguish $X^*$ from other random variables.
**Output :** A small set $\mathcal{C} \subseteq \mathcal{B}$ that contains $X^*$ with high probability.

```
for i ← 1 to |B| do
    σ_{X_i} ← 0;    // Storing the sum of samples from the random variable X_i.
for j ← 1 to s do
    for i ← 1 to |B| do
        Get a sample x_i from random variable X_i;
        σ_{X_i} ← σ_i + x_i;
return arg max{σ_{X_1}, σ_{X_2}, ..., σ_{X_n}};    // Set of X that has the largest sum.
```

**Theorem 1.** *There exists an algorithm with sample size $\mathcal{O}(\varepsilon^{-2} \log |\mathcal{B}|)$ that outputs a set $\mathcal{C} \subseteq \mathcal{B}$, such that $\mathcal{C} = \{X^*\}$ with probability at least $5/6$.*

The proof is a simple application of Hoeffding's bound.

*Proof.* Let $D_X := X - X^*$ be the difference between $X$ and $X^*$, where $X \in \mathcal{B} \backslash \{X^*\}$. We have $D_X \in \{-1, 0, 1\}$, and its expectation is

$$\mathsf{E}[D_X] = \mathsf{E}[X] - \mathsf{E}[X^*] = -\varepsilon.$$

Now, let $S_X := \sum_{i=1}^{s} D_X$, we have $\mathsf{E}[S_X] = s\,\mathsf{E}[D_X] = -\varepsilon s$. Using Hoeffding's inequality, we have

$$\Pr[S_X \geq 0] = \Pr[S_X - \mathsf{E}[S_X] \geq \varepsilon s] \leq \exp\left(-\frac{2\varepsilon^2 s^2}{4s}\right) = \exp\left(-\frac{1}{2}\varepsilon^2 s\right).$$

Therefore, the probability that $\mathcal{C} \neq \{X^*\}$ can be upper bounded using union bound,

$$\begin{aligned} \Pr[\mathcal{C} \neq \{X^*\}] &\leq \sum_{X \in \mathcal{B} \backslash \{X^*\}} \Pr[S_X \geq 0] \\ &\leq |\mathcal{B}| \exp\left(-\frac{1}{2}\varepsilon^2 s\right) = \frac{1}{6}, \end{aligned}$$

if we choose the sample size $s = 2\varepsilon^{-2} \log(6|\mathcal{B}|)$.

□

Using the median trick, we can prove a sample size of $\mathcal{O}(\varepsilon^{-2} \log |\mathcal{B}| \log \delta^{-1})$ so that $\mathcal{C}$ contains only $X^*$ with probability at least $1 - \delta$.

### 3.3.1 Sample Size for Mapper

To prove a suitable sample size for the mapper, we make the following assumptions:

1. $Q_{1,j}, Q_{2,j}, \ldots, Q_{|\mathcal{B}|,j}$ can be viewed as independent Bernoulli random variables. In other words, the $k$-mers contained in bucket $b_i$ is independent of that contained in $b_{i'}$, for each $i \neq i'$.

2. The pattern string can be mapped uniquely to one position in the genome, i.e. it doesn't come from a repeat region and can only be contained entirely in one and only one bucket $i^* \in \mathcal{B}$.

3. The pattern and the text are sufficiently close. In particular $\varepsilon > 1/4$, or the two strings share at least $1/4$ of all the $k$-mers.

Using the previous conclusions, we can show that choosing a sample size of $s = \mathcal{O}(\varepsilon^{-2} \log |\mathcal{B}|)$ and an error threshold of $e \leq s = \mathcal{O}(s)$ suffices to filter out almost all other false buckets with high probability.

**Corollary 1.** *Given pattern $P$, if the text $T$ contains a unique solution to the $(\varepsilon, k)$-matching problem, then we only need to sample $\mathcal{O}(\varepsilon^{-2} \log |\mathcal{B}|)$ $k$-mers from $P$ so that we can filter out only the correct bucket containing $P$ using Algorithm 1 with probability at least 5/6.*

*Proof.* For each bucket $i \in \mathcal{B} \backslash \{i^*\}$ that don't contain the read, there can be at most $(m - k + 1)$ $k$-mers, meaning that there can be at most $(m - k + 1)$ ones in the bit vector $Q_{i,\cdot}$, i.e. for all $i = 1, 2, \ldots, |\mathcal{B}|$,

$$\sum_{j=1}^{4^k} Q_{i,j} \leq m - k + 1 \leq m \Rightarrow \mathsf{E}[Q_{i,j}] = \Pr[Q_{i,j} = 1] \leq \frac{m}{4^k}.$$

On the other hand, the probability for the target bucket $i^*$ can be calculated using the law of total probability,

$$\begin{aligned} \mathsf{E}[Q_{i^*,j}] = \Pr[Q_{i^*,j} = 1] &= \Pr[Q_{i^*,j} = 1 \mid j \in \mathcal{M}_k(S, P)] \Pr[j \in \mathcal{M}_k(S, P)] \\ &\quad + \Pr[Q_{i^*,j} = 1 \mid j \notin \mathcal{M}_k(S, P)] \Pr[j \notin \mathcal{M}_k(S, P)] \\ &\geq 1 \cdot \varepsilon \end{aligned}$$

Therefore, we have

$$\Pr[Q_{i^*,j} = 1] - \Pr[Q_{i,j} = 1] \geq \varepsilon - \frac{m}{4^k} = \varepsilon - \frac{1}{4} = \Theta(\varepsilon)$$

if we choose bucket size $m = 4^{k-1}$. Now, by Theorem 1, we can conclude that a sample size of $\mathcal{O}(\varepsilon^{-2} \log |\mathcal{B}|)$ is enough so that the filter returns only $i^*$ with probability at least 5/6. □

Therefore, the total time used by the mapper is bounded by $\mathcal{O}(\varepsilon^{-4}|\mathcal{B}| \log^2 |\mathcal{B}|)$, which is independent of read length.

To use Hoeffding's inequality to upper bound the probability of failure, we assume that the random variable $D_X$ are independent of each other. Unfortunately, that might not be the case for the mapper, as $Q_{i,j_1}$ and $Q_{i,j_2}$ might be highly correlated, especially when the $k$-mers $j_1$ and $j_2$ are highly overlapped with each other. We talk about how we maximize the independence between the presence of sampled $k$-mers in the actual implementation in subsection 4.1.1.

### 3.3.2 Sample Size for Locator

**Corollary 2.** *Given pattern $P$, if the text $T$ of length $m$ contains a unique solution to the $(\varepsilon, k)$-matching problem, then we only need to sample $\mathcal{O}(\varepsilon^{-2} \log m)$ $k$-mers from $P$ so that we can output only the correct location using the locator with probability at least 5/6.*

The proof and assumptions are mostly similar to the previous proof, except that the set $\mathcal{B}$ now represents the set of all $\mathcal{O}(m)$ possible positions inside the bucket. Assuming $\mathcal{O}(1)$ time to query from the $k$-mer hash table, the total time complexity for the locator will be $\mathcal{O}(\varepsilon^{-2} \log m)$ on average, which is again independent of read length.

Finally, we can prove an upper bound for sample size that guarantees the high sensitivity of the BucketMap algorithm.

**Theorem 2** (Sensitivity of BucketMap)**.** *Given pattern $P$, if the text $T$ of length $n$ contains a unique solution to the $(\varepsilon, k)$-matching problem, then the BucketMap algorithm requires in total $\mathcal{O}(\varepsilon^{-2} \log n)$ $k$-mer samples from $P$ so that we can output only the correct location with probability at least 2/3.*

This result can be seen using a union bound with Corollary 1 and Corollary 2. Again, using the median trick, we can improve the sensitivity to be at least $1 - \delta$ with $\mathcal{O}(\varepsilon^{-2} \log n \log \delta^{-1})$ samples. Note that this theorem also ensures that the false positions are not returned by BucketMap with high probability. In other words, the specificity is also at least $1 - \delta$.

One thing to note is that if the pattern $P$ can be matched with a substring $S$ in $T$ exactly without any sequencing errors or mutations, or the quality filter successfully filters out all errors, then both the mapper and the locator must return the correct position as one of the outputs, and Bucketmap guarantees full sensitivity.

# Chapter 4

# Results

## 4.1 Implementation

To verify the efficiency of our algorithm, we develop a novel read mapper using C++ and the SeqAn 3 library [38]. The code as well as the benchmark experiments can be found under the repository https://github.com/GZHoffie/bucket-map. There are several details of implementation we want to discuss here:

### 4.1.1 Sampling

It is worth noticing that in the analysis of Corollary 1 and Corollary 2, we assume the independence between the presence of $k$-mers so that we can use Hoeffding's inequality to upper bound the probability of failure. However, it is generally not the case. For example, if the $k$-mer `ACCGTGA` is present in a bucket, then the $k$-mer `CCGTGAA` is likely to also be present in the bucket. The dependency of $k$-mers increases the probability of failure.

To weaken this dependency, we take two approaches.

1. Use of spaced $k$-mers [31]. Spaced $k$-mers are non-consecutive sequences of $k$ characters. Compared with the usual $k$-mers, which share $k-1$ characters with their neighbors, spaced $k$-mers generally share fewer characters with other seeds, which weakens the dependency and guarantees high sensitivity.

2. Evenly-spaced sampling. Instead of randomly sampling from all the $k$-mers, we choose the $k$-mers so that the space between them is roughly equal. This

is to ensure that the overlap between each pair of sampled $k$-mer is minimized.

### 4.1.2 Choice of Seed and Bucket Size

In the experiments, we choose the size of the seeds $k = 9$ and the bucket size $m = 4^{k-1} = 65536$. This choice is based on our observation that choosing a smaller $k$ will lead to a large number of buckets, making the mapping phase slow; while a larger choice of $k$ will make our bucket larger, making it inaccurate when determining the exact position of reads within each bucket. The choice of $k = 9$ is good for both shorter plant genomes and longer human genomes.

In our experiments, we choose to use ungapped 9-mers for both the mapper and locator. The choice of seed templates is purely random, as we don't observe a large difference in performance between spaced and ungapped seeds. However, the seeds are preferred to be not too long so that the effects of mismatches and indels are minimized.

## 4.2 Benchmark

We primarily test the performance of BucketMap in the case of Next Generation Sequencing, i.e. shorter read length and lower error rates.

### 4.2.1 Datasets

We use both simulated reads and real Illumina read datasets to test BucketMap on 3 different reference genomes.

#### Reference Genome

Supported by Singapore Wilmar International Ltd, we primarily use the reference and read data of *Elaeis guineensis* (EGU), a kind of palm tree that originated in West Africa and used for producing palm oil [35].

As this reference genome is not publicly available, we also use the other publicly available dataset: The human reference genome sequence GRCh38 from NCBI, available at https://www.ncbi.nlm.nih.gov/genome/guide/human/, and the *Escherichia Coli* strain K-12 substrain MG1655 assembly from ENA, available at https:

//www.ebi.ac.uk/ena/browser/view/GCA_000005845.2. For the GRCh38 dataset, we delete the ambiguous characters "N" at the beginning of the reference genome.

Table 4.1 is a summary of the information on reference genomes we use in the experiments.

**Table 4.1:** Information of reference genomes.

| Abbreviation | Species | Number of Bases | Number of Sequences |
|---|---|---|---|
| EColi | *Escherichia Coli* | 4,641,652 | 1 |
| EGU | *Elaeis Guineensis* | 1,701,312,507 | 932 |
| GRCh38 | *Homo Sapiens* | 3,136,819,257 | 705 |

### Simulated Reads

To mimic the behavior of DNA read sequencing machines, we implement a simple simulator that generates sequencing data. The users are allowed to set 5 parameters, as shown in Table 4.2.

**Table 4.2:** Parameters of DNA read simulator.

| Parameters | Meaning |
|---|---|
| $N$ | Number of reads in the generated `.fastq` file |
| $r$ | Length of the reads |
| $s$ | Substitution rate (number of substitutions per base sequenced) |
| $i$ | Insertion rate (number of substitutions per base sequenced) |
| $d$ | Deletion rate (number of deletions per base sequenced) |

The simulator, given the parameters, is able to generate $N$ sequenced reads of length close to $r$. The simulator first samples strings of length $r$ uniformly across all chromosomes and contigs in the reference genome (`.fasta` file). Then, it adds sequencing errors to the reads. The number of substitutions, for example, is sampled from a Poisson distribution with parameter $\lambda = s \cdot r$. The same goes for insertions and deletions. With a 50% chance, we record the reverse complement of the sampled read to mimic sequencing on the reverse strand.

We primarily consider the sequencing errors, as the rate of mutations, which is around $3 \times 10^{-5}$ mutations per base pair per cell generation [4], is much smaller than the sequencing error rates.

We use the dataset generator to simulate Illumina reads, and choose a mismatch rate of 0.2% and an indel rate of 0.05%. This is a reasonable choice of parameters, as the mean error rate is only 0.21% for R1 reads according to the Illumina error profile [40], [43], and an indel rate of 0.01% is a widely used value of read simulators and benchmark datasets [17], [18].

#### Real Reads

Apart from simulated datasets, we also test the performance of the mappers on real Illumina read datasets from ENA.

We use four datasets, two simulated and two real datasets, to test the mappers in a comprehensive manner. Table 4.3 shows a summary of the information of datasets we are testing.

**Table 4.3:** Information of datasets used in our experiments. The datasets S1 and S2 are using simulated reads, while R1 and R2 are using real reads coming from Illumina experiments.

| Dataset | Reference | Reads | # Sequences | # bases |
|---|---|---|---|---|
| S1 | EGU | Simulated | 1,000,000 | 300,000,103 |
| S2 | GRCh38 | Simulated | 1,000,000 | 300,000,212 |
| R1 | EColi | DRR035999 | 1,302,395 | 352,233,287 |
| R2 | EGU | TS1.81.90.001 | 4,922,564 | 669,468,704 |

### 4.2.2 Metrics

We compare the read mapping tools on both the time and memory usage and the accuracy to have a comprehensive view of how well the tools perform.

#### Time and Memory Usage

We test the speed and memory efficiency using the `/usr/bin/time -v` command in Linux [6], and collect the following fields from the output for analysis,

- `User time` ("Time (s)" in the tables): the total time used by the process (in user space), in seconds.
- `Maximum resident set size` ("Max RSS (Kb)" in the tables): the peak memory usage of the process, in Kilobytes.

- `Major (requiring I/O) page faults` ("#Major PF" in the tables): number of major page faults (reading from disk) while the process was running.
- `Minor (reclaiming a frame) page faults` ("#Minor PF" in the tables): number of minor page faults (updating pages) while the process was running.

**Accuracy**

The accuracy performance is done by comparing the output `.sam` files against the ground truth in the simulated dataset, where the following metrics are used.

- `Percentage of reads mapped` ("%Mapped" in the tables): Percentage of the reads that are mapped to at least 1 location.
- `Number of mapped locations per mapped read` ("Avg. #Map Loc." in the tables): Average number of locations returned for each mapped read in the `.sam` file.
- `Percentage of reads mapped correctly` ("Sensitivity" in the tables): Percentage of the reads that are mapped to the correct location in the reference genome. This measures how many reads are we able to correctly utilize from the dataset.
- `Percentage of mapped positions being correct` ("Precision" in the tables): Percentage of mapped locations returned by the mapping algorithm that are correct. This measures the quality of returned mappings.

If we are using a simulated dataset, a mapped location is defined to be "correct" if it matches the true location where we sample the read sequence. In the case of real datasets, it is considered "correct" if it matches the returned results of Bowtie2 or BWA-MEM. We regard the results of these two mappers as correct answers because they are the most well-known and widely-used, and display both high sensitivity and high precision in the simulated datasets.

### 4.2.3 Results

The experiments are run on a personal computer with 8 cores, 16 logical processors, and 16.0 Gb memory. The CPUs are Intel® Core™ i7-10875H with a clock speed of 2.30 GHz and an L1 cache size of 512 Kb. The running of the mapping programs and benchmarking is done on Windows Subsystem for Linux (WSL2).

We benchmark BucketMap against the following popular mappers,

- Bowtie2 [25],
- BWA-MEM [29],
- Subread [32],
- Minimap2 [28].

Bowtie2 and BWA-MEM are the popular tools using FM-index, Subread is the state-of-the-art mapper that is also using the seed-and-vote scheme (similar to the locator of BucketMap), and Minimap2 is the state-of-the-art general-purpose alignment program.

All mappers are run under the default settings, under the single-threaded version so that the percentage of CPU a job gets doesn't exceed 100% for all mappers (except for Minimap2 since it is slow in the single-threaded version).

#### Indexing

Since each mapper is using a different indexing strategy, and indexing is only run once for each reference genome, a benchmark for the indexing stage is less important. However, it might be a good reference for users that need to perform the mapping on several different references, or those who only want to map a few reads.

As shown in Table 4.4, the first two tools, which are using FM-index are taking longer time to build the index files than the other three mappers. This is mainly due to the need to sort all the rotations of the reference genome in lexicographic order, which takes $\mathcal{O}(n \log n)$ time.

In particular, our mapper proved up to 10 times faster than the mappers using FM-index and 4 times faster than its seed-and-vote counterpart. Moreover, running

**Table 4.4:** Time & Memory performance of different mappers during indexing on the EGU reference genome. The last column corresponds to the total disk usage (DU) of all the index files produced by each tool.

| Tool | Time (s) | Max RSS (Kb) | #Major PF | #Minor PF | DU (Kb) |
|---|---|---|---|---|---|
| Bowtie2 | 2446.30 | 7,648,524 | 832,483 | 7,510,117 | 2,361,392 |
| BWA-MEM | 1545.29 | 2,495,668 | 13 | 424,721 | 2,907,664 |
| Subread | 684.65 | 3,197,744 | 8 | 1,347,347 | 3,076,680 |
| Minimap2 | 68.22 | 6,868,580 | 1,121 | 3,067,356 | 4,040,908 |
| BucketMap | 159.72 | 1,151,240 | 41 | 10,903 | 848,556 |

indexing is using up to 1.1 Gb of memory, and the resultant index file takes only 0.8 Gb of disk space. Both the time and memory usage are only linearly dependent on the length of the reference genome, which is lightweight and acceptable even for users with lower-end computers.

### Mapping and Alignment Verification

To measure the performance of mapping, we take both speed and accuracy into account. We benchmark the performance of all mappers along with BucketMap. The time and memory performance are summarized in Table 4.5, while the accuracy benchmark is shown in Table 4.6.

**BucketMap is fast, especially for shorter reference genomes and longer reads.** In datasets S1 and R1, where the lengths of the reference are short, BucketMap is 2-3 times faster than all the other state-of-the-art methods. However, when it comes to longer references such as GRCh38, BucketMap is slower than its counterparts. This is mainly because of the longer bit vectors in the mapper function and the need to create $k$-mer index hash tables during the mapping, which will be discussed in subsection 4.3.1.

**BucketMap is extremely memory-saving and cache-efficient.** Due to the hierarchical mapping strategy of BucketMap, there is no need to keep one, large index file in memory when finding the exact locations of the read. As a result, BucketMap is using 2-6 times less memory than all other state-of-the-art mappers, making it extremely lightweight and easy to use for users with different computer setups. Moreover, the number of major and minor page faults is relatively small for all datasets, which further proves that BucketMap is memory- and cache-efficient.

**Table 4.5:** Time & Memory performance of different mappers during mapping on simulated and real datasets. The description of datasets is shown in Table 4.3. When running dataset S2, Bowtie2 fails to build a valid index file, while Minimap2 fails to output a valid `sam` file on my computer, possibly because of the low memory of my virtual machine.

| Dataset | Tool | Time (s) | Max RSS (Kb) | #Major PF | #Minor PF |
|---|---|---|---|---|---|
| S1 | Bowtie2 | 924.36 | 1,967,648 | 12 | 247,445 |
| | BWA-MEM | 501.49 | 2,995,216 | 6 | 615,752 |
| | Minimap2 | 1407.40 | 6,312,564 | 15 | 2,989,707 |
| | Subread | 506.99 | 4,198,412 | 2768 | 773,325 |
| | BucketMap | **278.89** | **881,180** | 0 | 220,659 |
| S2 | Bowtie2 | - | - | - | - |
| | BWA-MEM | 457.32 | 5,448,508 | 7 | 1,086,184 |
| | Minimap2 | - | - | - | - |
| | Subread | **303.60** | 6,592,860 | 60,714 | 1,522,748 |
| | BucketMap | 521.22 | **1,575,680** | 1 | 273,215 |
| R1 | Bowtie2 | 196.40 | **30,520** | 0 | 6,632 |
| | BWA-MEM | 127.84 | 104,152 | 0 | 25,688 |
| | Minimap2 | 103.88 | 1,094,160 | 1 | 254,514 |
| | Subread | 90.94 | 1,171,212 | 1 | 75,180 |
| | BucketMap | **27.14** | 170,116 | 0 | 17,684 |
| R2 | Bowtie2 | 743.00 | 1,961,608 | 115 | 8,232 |
| | BWA-MEM | 771.18 | 3,020,384 | 8 | 214,991 |
| | Minimap2 | 411.43 | 7,149,336 | 329,494 | 2,304,936 |
| | Subread | **256.91** | 4,942,176 | 1,046 | 926,052 |
| | BucketMap | 576.08 | **1,081,480** | 12 | 192,779 |

One exception happens when the reference genome is very small (e.g. dataset R1). In this case, index files of all mappers are small and take up very little memory. BucketMap is using a bit more memory than the other mappers due to the need to store sampled $k$-mers from all the reads.

**BucketMap guarantees high sensitivity, even for longer reference genomes with repetitive regions.** As shown in Table 4.6, BucketMap maintains a high mapping rate of above 98% for all simulated datasets and close to 90% for real datasets. For the EColi and EGU references, BucketMap is comparable to the FM-index-based methods. For the longer GRCh38 genome which contains more repetitive regions, our mapper beats other mappers with a sensitivity of 97.5%. The results show that BucketMap has the potential to efficiently and correctly identify

**Table 4.6:** Accuracy performance of different mappers during mapping on simulated and real datasets. The description of datasets is the same as Table 4.5. For real datasets R1 and R2, sensitivity and precision are measured with respect to the returned results of Bowtie2 and BWA-MEM, as discussed in subsection 4.2.1, therefore they are having a 100% precision.

| Dataset | Tool | %Mapped | Avg. #Map Loc. | Sensitivity | Precision |
|---|---|---|---|---|---|
| S1 | Bowtie2 | 99.89% | 1.000 | 97.64% | 97.75% |
| | BWA-MEM | **100.00%** | 1.001 | 98.10% | 97.96% |
| | Minimap2 | 99.70% | 1.396 | **99.23%** | 71.27% |
| | Subread | 96.74% | 1.000 | 94.94% | **98.14%** |
| | BucketMap | 98.95% | 1.149 | 97.50% | 85.74% |
| S2 | Bowtie2 | - | - | - | - |
| | BWA-MEM | **100.00%** | 1.000 | 92.61% | 92.61% |
| | Minimap2 | - | - | - | - |
| | Subread | 88.17% | 1.000 | 86.53% | **98.14%** |
| | BucketMap | 99.17% | 1.355 | **97.42%** | 72.51% |
| R1 | Bowtie2 | 81.66% | 1.000 | *81.66%* | *100.00%* |
| | BWA-MEM | **89.75%** | 1.127 | *89.75%* | *100.00%* |
| | Minimap2 | 89.20% | 1.196 | **86.05%** | 88.12% |
| | Subread | 85.29% | 1.000 | 77.78% | **91.19%** |
| | BucketMap | 89.62% | 1.077 | 82.60% | 86.70% |
| R2 | Bowtie2 | 90.61% | 1.000 | *90.61%* | *100.00%* |
| | BWA-MEM | **96.86%** | 1.022 | *96.86%* | *100.00%* |
| | Minimap2 | 94.12% | 1.265 | **93.41%** | 80.72% |
| | Subread | 83.51% | 1.000 | 81.49% | **97.58%** |
| | BucketMap | 89.53% | 1.066 | 86.36% | 91.77% |

the correct locations of reads, even for long and highly-repetitive reference genomes.

**The alignment-free locator of BucketMap guarantees high mapping quality.** For the reference genome of EColi and EGU, BucketMap guarantees a high precision of at least 85%, which is comparable to Minimap2 and shows that the locator is able to filter out incorrect alignments, even with the absence of dynamic programming algorithms for alignment verification.

To sum up, the results on the simulated datasets show that BucketMap is lightweight and memory-efficient, guarantees high speed and sensitivity, and easily scales up to process long human genomes and longer reads efficiently and accurately.

## 4.3 Discussion

In this section, we discuss some limitations to the current implementation of BucketMap, as well as its time and memory bottleneck.

### 4.3.1 Time Usage of BucketMap

To find the bottleneck of time usage of BucketMap, we evaluate the time usage of during the mapping of dataset S1 (see Table 4.5).

**Table 4.7:** Time usage by different components of BucketMap during mapping reads in dataset D2. Miscellaneous parts contain reading query files and the reference genome, etc.

| Tasks | Time (s) | %time usage |
|---|---|---|
| Loading matrix $Q$ | 14.936 | 5.33% |
| Mapper | 89.48 | 31.92% |
| Building $k$-mer index | **121.984** | **43.52%** |
| Locator | 15.7 | 5.60% |
| Miscellaneous | 38.18 | 13.60% |
| **Total** | 280.28 | 100% |

Around 50% of the running time is used to build $k$-mer index hash tables during the locator phase. For each bucket, we read the corresponding part of the reference genome and build a hash table that maps a $k$-mer to its positions inside the bucket. We choose to build the hash tables on the fly during the mapping instead of during the indexing phase. This is because the hash tables are large, taking at least 200 Kb each, and tens of gigabytes in total. It would be slow to store and load those hash tables from disks. This is, indeed, a flaw in the current implementation which can be improved further.

Luckily, the time needed for $k$-mer index building is a constant for a given reference genome and is not going to increase a lot as the number of reads increases.

### 4.3.2 Memory Usage of BucketMap

We also evaluate the memory usage during the run of BucketMap using the `massif` tool, as shown in Figure 4.1. Apart from the initializing phase, the peak memory usage (846 Mb) happens during the mapping phase, where we have to

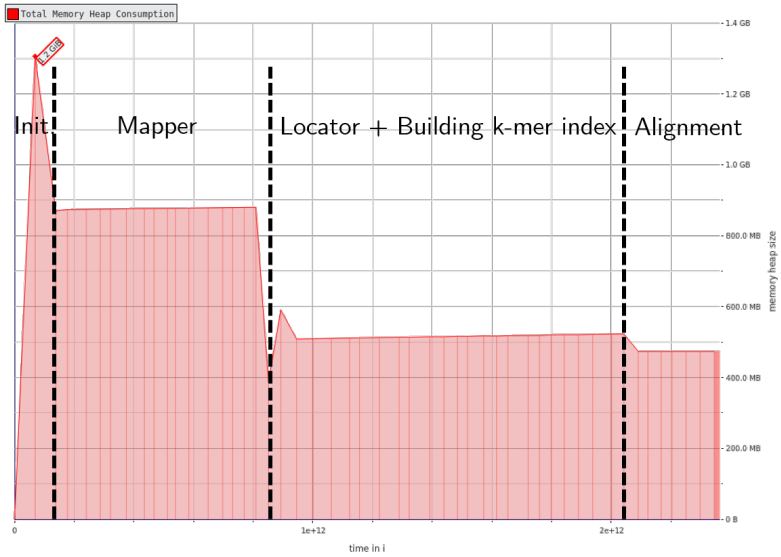


**Figure 4.1:** The memory usage during the run of BucketMap process (for dataset S1).

store the bit matrix $Q$ of size $|\mathcal{B}| \times 4^k$. Since we choose the bucket size to be $m = 4^{k-1}$, the size of $Q$ will be $|\mathcal{B}| \times 4^k = \lceil n/m \rceil \times 4^k = \mathcal{O}(n)$, which will be linearly dependent on the size of the reference genome.

As the number of reads grows in the sequence file (comparing datasets S1 and R2 which are both using EGU as reference), we also observe an increase in memory usage. This is because the current implementation stores all reads in the memory for faster querying. The details are discussed in subsection 4.3.3.

### 4.3.3 Current Limitations of BucketMap

There are currently several requirements for the datasets that are given to BucketMap for it to outperform current methods:

1. Low error rate $e$. As proved in section 3.3, the sample size and running time of our algorithm are linearly dependent on $\varepsilon^{-2}$, where $\varepsilon$ is the percentage of $k$-mers that can be matched between the pattern and the text. Assuming that errors happen independently, we have

$$\mathrm{E}[\varepsilon] = (1 - e)^k,$$

In the case of Nanopore sequencing [11], an error rate of 7.5% and $k = 9$ leads to a low $k$-matching rate of $\mathrm{E}[\varepsilon] = 0.495 \approx 1/2$. This means that the sample

size needs to be at least 4 times more than the case in Illumina sequencing to maintain the same sensitivity, making our algorithm inefficient. In conclusion, BucketMap works best for reads with an error rate of less than 2%. A more clever way of sampling is needed for efficient mapping in Third Generation Sequencing.

2. Small number of reads per run. As discussed in subsection 3.2.4, we store the $k$-mers sampled from all reads in the memory to avoid reading the sequence file multiple times. This approach might be inefficient if the number of reads goes large. In particular, the space needed to store the reads is linearly dependent on the number of reads (about 57 Mb per 1M reads) and grows large for a larger query file, increasing memory usage and causing a lot of cache misses when accessing the reads. As a result, BucketMap works best for the case where the number of reads is small (under 5 million), and it is recommended that larger sequence files are broken down into smaller ones for cache efficiency. This might be solvable if BucketMap is implemented under distributed computing paradigm, which will be discussed in subsection 5.2.1.

In a nutshell, the current implementation of BucketMap works best with a short reference genome, a low error rate, and a small number of reads.

# Chapter 5

# Conclusion and Future Work

## 5.1 Conclusion

The recent development of Next- and Third-generation sequencing brings a tremendous growth of DNA read databases. Being the first step of data analysis, the problem of mapping short DNA reads to their positions in the long reference genome accurately and efficiently is increasingly crucial. The current mappers, though already quite fast and accurate, are still time- and memory-inefficient due to the use of large index files and slow alignment verification.

We defined a new definition of the goodness of alignment, $(\varepsilon, k)$-matching, which is intuitive and biologically meaningful. Then, we proposed a novel mapping algorithm, *BucketMap*, that finds the solutions to the $(\varepsilon, k)$-matching problem in $\mathcal{O}(\varepsilon^{-4}|\mathcal{B}| \log^2 |\mathcal{B}| \log \delta^{-1} + \varepsilon^{-2} \log m \log \delta^{-1})$ time to achieve a sensitivity and specificity of at least $1 - \delta$, which is independent of read length, proving potential application in long read mapping.

BucketMap adopts a hierarchical mapping strategy: divide the reference genome into smaller buckets, map reads to their candidate buckets, and then find their exact locations within the candidate buckets. With this strategy, we only need to keep a small index file for one bucket at a time, greatly reducing the memory required for mapping. In our experiments, BucketMap proved to be using only 1.5 Gb of memory to map 1 million reads to the human reference genome, which is 2-6 times less memory than the other state-of-the-art mappers.

BucketMap also utilizes $k$-mer sampling together with hashing and the seed-and-vote paradigm to achieve sub-linear time in both the mapper and the locator. It

proved to be especially efficient for shorter reference genomes and longer reads, being 2-3 times faster than the other mappers. With the help of our novel distinguishability filter and quality filter, BucketMap also maintains high sensitivity and precision for all the datasets, while beating Subread, its seed-and-vote counterpart.

The performance of BucketMap proves the possibility of utilizing ideas of randomized and cache-oblivious algorithms in the DNA read mapping problem.

## 5.2 Open Questions

In this final section, we discuss some major open questions, as well as ways to the improvement of BucketMap and points for future research.

### 5.2.1 Potential Improvements for BucketMap

There are various ways of potentially improving the sensitivity and speed of our mapper.

#### Sampling of $k$-mers

In this paper, we proposed a total sample size of $\mathcal{O}(\varepsilon^{-2} \log n \log \delta^{-1})$ to ensure a sensitivity of at least $1 - \delta$. However, this sample size might not be optimal. It might be meaningful to prove the optimality of this sample size or propose a more clever way of sampling high-quality and error-free $k$-mers.

Apart from improving the number of $k$-mer samples, we can also attempt to try out some different shapes of the $k$-mer seeds. Currently, we only choose one shape for the spaced seed in the mapper and the locator. It might be interesting to try with a set of different spaced seeds [19], and see how the performance varies.

#### SIMD Parallelism and Multi-threading

The current implementation of BucketMap is single-threaded and has much room for improvement in speed. Currently, the speed of the mapper is dependent on the choice of sample size $s$, the maximum number of missed $k$-mers $e$, and the number of buckets $|\mathcal{B}|$. The procedure of the bucket filter (Algorithm 1) can be parallelized by maintaining multiple instances of bucket filters, each handling a different, smaller

group of sampled $k$-mers and a smaller error threshold $e$. To filter out the correct bucket, we can just take the bitwise majority of all boolean vectors $F_e$. In this way, we can minimize $s$ and $e$ for each processor, and allow a higher speed for the mapper.

The implementation of the locator can also speed up using multi-threading. In the locator, the task for each bucket is disjoint: we build a separate $k$-mer index hash table and map a separate set of reads for each bucket. This indicates that we can have one thread dealing with one bucket at a time, which will greatly improve the time performance.

#### Batch Processing of Buckets

For longer reference genomes such as GRCh38, our locator algorithm suffers due to a larger number of buckets. Under the current implementation, for each bucket, we go through all reads that are mapped to it. This implies that we need to go through all the reads $|\mathcal{B}|$ times, which is very inefficient as $|\mathcal{B}|$ goes large. Batch processing of the buckets may greatly improve the speed. If we process $c$ buckets at a time, for example, the number of times we go through the reads would reduce to $|\mathcal{B}|/c$, while still keeping a small memory usage.

#### Distributed Read Mapping

The hierarchical algorithm can be easily integrated into distributed computing paradigms, such as MapReduce applications [10]. In particular, The mapper node uses BucketMap's mapper function and distributes the reads to the other nodes in the cluster, each handling a subset of buckets. Under this setting, there would be no need to store all reads in the memory of the mapper node anymore, allowing the mapping of a huge number of reads.

### 5.2.2 Potential Applications

The idea of hierarchical mapping, as well as the $k$-mer sampling technique, can potentially be helpful for solving other interesting problems in Computational Biology, or even other fields of Computer Science.

### Faster Detection of Structural Variations

As our mapper that maps read to different buckets is fast, we can potentially use the mapping algorithm to identify structural variations. If a (paired-end) read is assigned multiple buckets, but the underlying $k$-mers appear to split into two groups such the buckets associated with these two groups are disjoint, then it can possibly indicate a structural variation where a segment of the genome is transposed to another location.

### Faster Database Querying

The hierarchical mapping strategy can also be applied to other problems such as protein function prediction. One of the traditional methods of predicting the function of a protein is to find proteins in the database that are similar to it. This can be done by finding the common $k$-mers between the proteins [47], which is very similar to the idea of mapper in BucketMap, indicating that the bit-parallel algorithms and data structures of BucketMap might be useful to improve the speed and sensitivity of this process.

### Mapping to Multiple Genomes

The algorithms used by the mapper and locator in BucketMap allow mapping to multiple reference genomes for more accurate single nucleotide polymorphisms (SNPs) detection and phylogenetic inference [45]. The procedure may look like the following:

1. Align the whole reference genomes using whole-genome alignment (WGA) methods.
2. Divide the alignment into buckets, where each bucket contains the same region of all the genomes.
3. Build the index matrix $Q$ using $k$-mers from all the genomes in each bucket.

The rest would be the same as our BucketMap procedure. This idea might be useful for trio-sequencing, where we can map reads from a child's DNA to both paternal and maternal genomes. It might also be potentially useful for metagenome analysis, allowing fast taxonomic annotation [44].

We believe the above (but not limited to) future research directions will advance the technology presented in this thesis and contribute to academia and industry.

# Bibliography

[1] J. Acharya, C. Daskalakis, and G. Kamath, "Optimal testing for properties of distributions", *Advances in Neural Information Processing Systems*, vol. 28, 2015.

[2] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, *et al.*, "Personalized copy number and segmental duplication maps using next-generation sequencing", *Nature genetics*, vol. 41, no. 10, pp. 1061–1067, 2009.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool", *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[4] S. J. Balin and M. Cascalho, "The rate of mutation of a single gene", *Nucleic Acids Research*, vol. 38, no. 5, pp. 1575–1582, 2010, PMID: 20007603 PMCID: PMC2836558. DOI: 10.1093/nar/gkp1119. [Online]. Available: https://doi.org/10.1093/nar/gkp1119.

[5] G. Benson, A. Levy, S. Maimoni, D. Noifeld, and B. R. Shalom, "Lcsk: A refined similarity measure", *Theoretical Computer Science*, vol. 638, pp. 11–26, 2016.

[6] A. Brouwer, "Time(1) linux manual page", 2000. [Online]. Available: https://man7.org/linux/man-pages/man1/time.1.html.

[7] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron, "Q-gram based database searching using a suffix array (quasar)", in *Proceedings of the third annual international conference on Computational molecular biology*, 1999, pp. 77–83.

[8] M. Burrows and D. Wheeler, "A block-sorting lossless data compression algorithm", *Technical Report 124, Digital SRC Research Report*, vol. 2089, 1994.

[9] S. Canzar and S. L. Salzberg, "Short read mapping: An algorithmic tour", *Proceedings of the IEEE*, vol. 105, no. 3, pp. 436–458, 2015.

[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters", *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] C. Delahaye and J. Nicolas, "Sequencing dna with nanopores: Troubles and biases", *PloS one*, vol. 16, no. 10, e0257521, 2021.

[12] P. Ferragina and G. Manzini, "Opportunistic data structures with applications", in *Proceedings 41st annual symposium on foundations of computer science*, IEEE, 2000, pp. 390–398.

[13] P. Ferragina and G. Manzini, "Indexing compressed text", *Journal of the ACM (JACM)*, vol. 52, no. 4, pp. 552–581, 2005.

[14] O. Gotoh, "An improved algorithm for matching biological sequences", *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, 1982.

[15] P. Guan and W.-K. Sung, "Structural variation detection using next-generation sequencing data: A comparative technical review", *Methods*, vol. 102, pp. 36–49, 2016.

[16] F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp, "Mrsfast: A cache-oblivious algorithm for short-read mapping", *Nature methods*, vol. 7, no. 8, pp. 576–577, 2010.

[17] A. Hatem, D. Bozda, A. E. Toland, and Ü. V. Çatalyürek, "Benchmarking short sequence mapping tools", *BMC bioinformatics*, vol. 14, pp. 1–25, 2013.

[18] M. Holtgrewe, "Mason: A read simulator for second generation sequencing data",, 2010.

[19] L. Ilie and S. Ilie, "Multiple spaced seeds for homology search", *Bioinformatics*, vol. 23, no. 22, pp. 2969–2977, 2007.

[20] "Illumina sequencing platforms", 2023. [Online]. Available: https://www.illumina.com/systems/sequencing-platforms.html.

[21] J. S. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies", *BMC genomics*, vol. 19, no. 2, pp. 23–40, 2018.

[22] D. C. Koboldt, "Best practices for variant calling in clinical sequencing", *Genome Medicine*, vol. 12, no. 1, pp. 1–13, 2020.

[23] K. Krishnamachari, D. Lu, A. Swift-Scott, A. Yeraliyev, K. Lee, W. Huang, S. N. Leng, and A. J. Skanderup, "Accurate somatic variant detection using weakly supervised deep learning", *Nature Communications*, vol. 13, no. 1, pp. 1–8, 2022.

[24] G. M. Landau, U. Vishkin, and R. Nussinov, "An efficient string matching algorithm with k differences for nucleotide and amino acid sequences", *Nucleic acids research*, vol. 14, no. 1, pp. 31–46, 1986.

[25] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2", *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.

[26] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome", *Genome biology*, vol. 10, no. 3, pp. 1–10, 2009.

[27] V. Levenshtein, "Binary codes capable of correcting spurious insertions and deletions of ones", *Russian Problemy Peredachi Informatsii*, vol. 1, pp. 12–25, 1965.

[28] H. Li, "Minimap2: Pairwise alignment for nucleotide sequences", *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[29] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform", *bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[30] H. Li, J. Ruan, and R. Durbin, "Mapping short dna sequencing reads and calling variants using mapping quality scores", *Genome research*, vol. 18, no. 11, pp. 1851–1858, 2008.

[31] M. Li, B. Ma, D. Kisman, and J. Tromp, "Patternhunter ii: Highly sensitive and fast homology search", *Journal of bioinformatics and computational biology*, vol. 2, no. 03, pp. 417–439, 2004.

[32] Y. Liao, G. K. Smyth, and W. Shi, "The subread aligner: Fast, accurate and scalable read mapping by seed-and-vote", *Nucleic acids research*, vol. 41, no. 10, e108–e108, 2013.

[33] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches", *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[34] S. Nurk, S. Koren, A. Rhie, M. Rautiainen, A. V. Bzikadze, A. Mikheenko, M. R. Vollger, N. Altemose, L. Uralsky, A. Gershman, *et al.*, "The complete sequence of a human genome", *Science*, vol. 376, no. 6588, pp. 44–53, 2022.

[35] "Oil Palm Plantation & Milling - Wilmar International", 2021. [Online]. Available: https://www.wilmar-international.com/our-businesses/plantation/oil-palm-plantation-milling.

[36] "Quality scores for next-generation sequencing", Illumina, Inc., Tech. Rep., 2011. [Online]. Available: https://www.illumina.com/documents/products/technotes/technote_Q-Scores.pdf.

[37] K. R. Rasmussen, J. Stoye, and E. W. Myers, "Efficient q-gram filters for finding all $\epsilon$-matches over a given length", in *Research in Computational Molecular Biology: 9th Annual International Conference, RECOMB 2005, Cambridge, MA, USA, May 14-18, 2005. Proceedings 9*, Springer, 2005, pp. 189–203.

[38] K. Reinert, T. H. Dadi, M. Ehrhardt, H. Hauswedell, S. Mehringer, R. Rahn, J. Kim, C. Pockrandt, J. Winkler, E. Siragusa, G. Urgese, and D. Weese, "The seqan c++ template library for efficient sequence analysis: A resource for programmers", *Journal of Biotechnology*, vol. 261, pp. 157–168, Nov. 2017. [Online]. Available: http://publications.imp.fu-berlin.de/2103/.

[39] D. Sankoff and J. B. Kruskal, "Time warps, string edits, and macromolecules: The theory and practice of sequence comparison", *Reading: Addison-Wesley Publication*, 1983.

[40] M. Schirmer, R. DAmore, U. Z. Ijaz, N. Hall, and C. Quince, "Illumina error profiles: Resolving fine-scale variation in metagenomic sequencing data", *BMC bioinformatics*, vol. 17, pp. 1–15, 2016.

[41] A. D. Smith, Z. Xuan, and M. Q. Zhang, "Using quality scores and longer reads improves accuracy of solexa read mapping", *BMC bioinformatics*, vol. 9, no. 1, pp. 1–8, 2008.

[42] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences", *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[43] N. Stoler and A. Nekrutenko, "Sequencing error profiles of illumina sequencing instruments", *NAR genomics and bioinformatics*, vol. 3, no. 1, lqab019, 2021.

[44] J. Tamames, M. Cobo-Simón, and F. Puente-Sánchez, "Assessing the performance of different approaches for functional and taxonomic annotation of metagenomes", *BMC genomics*, vol. 20, no. 1, pp. 1–16, 2019.

[45] C. Valiente-Mullor, B. Beamud, I. Ansari, C. Francés-Cuesta, N. Garca-González, L. Meja, P. Ruiz-Hueso, and F. González-Candelas, "One is not enough: On the effects of reference genome for the mapping and subsequent analyses of short-reads", *PLOS Computational Biology*, vol. 17, no. 1, e1008678, 2021.

[46] M. Vasimuddin, S. Misra, H. Li, and S. Aluru, "Efficient architecture-aware acceleration of bwa-mem for multicore systems", in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2019, pp. 314–324.

[47] S. Vinga and J. Almeida, "Alignment-free sequence comparisona review", *Bioinformatics*, vol. 19, no. 4, pp. 513–523, 2003.

[48] D. Weese, M. Holtgrewe, and K. Reinert, "Razers 3: Faster, fully sensitive read mapping", *Bioinformatics*, vol. 28, no. 20, pp. 2592–2599, 2012.

[49] K. A. Wetterstrand, "The cost of sequencing a human genome", 2021. [Online]. Available: https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost.

[50] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with fasthash", in *BMC genomics*, Springer, vol. 14, 2013, pp. 1–13.