ACCELERATING K-MER COUNTING FOR GENOMIC ANALYSIS

Gunavaran Brihadiswaran

(208032P)

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

June 2021

DECLARATION

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to the University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other media. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: *UOM Verified Signature* Date: 10.10.2021

The above candidate has carried out research for the Master's thesis under my supervision.

Name of the supervisor: Prof. V. S. D. Jayasena

Signature of the superviso *UOM Verified Signature* Date: 13/10/2021

Abstract

k-mer counting is the process of counting *k* length substrings in a sequence. It is an important step in many bioinformatics applications including genome assembly, sequence error correction, and sequence alignment. Even though generating *k*-mer histograms seems simple and straightforward, processing large datasets efficiently with limited resources, especially memory, is very challenging. As the advancements in next-generation sequencing technologies have resulted in a tremendous growth of genomic data, it is inevitable for *k*-mer counters to be faster and more efficient. A lot of work has been done in the past decade to optimize *k*-mer counting.

Frigate, a fast and efficient tool capable of counting and querying *k*-mers is presented. Its in-memory design utilizes multithreaded, lock-free data structures to improve performance. Thread synchronization is handled using the compare-and-swap technique. The parallel processing pipeline of Frigate is the result of careful performance engineering and design. Frigate was developed with the emphasis on values of *k* less than 20, aiming to maximize performance by employing different algorithms for different ranges of *k* values.

The performance of Frigate was compared with six state-of-the-art *k*-mer counters: Jellyfish, DSK, Gerbil, CHTKC, KMC2, and KMC3, using two real-world datasets. The experiments were carried out for *k* values of 10, 15, and 17 using a different number of threads in the range [1, 32]. The results show that Frigate achieves a comparable performance or up to 2-3x speedup compared to its competitors, especially for large datasets. The *k*-mer counters were analyzed based on the running time, amount of memory used, and scalability. The correctness of Frigate was evaluated by comparing the *k*-mer frequency histogram with those of other *k*-mer counters.

Frigate is written in C and freely available at https://github.com/Gunavaran/frigate under MIT license.

Keywords: *K*-mer counting, Genome analysis, Performance engineering, Parallel computing

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my supervisor Prof. Sanath Jayasena for giving me the opportunity to be involved in this research and for his invaluable support and guidance throughout this research. I am always thankful for his efforts and kindness when we were struggling to acquire research grants.

I am also thankful to Systems Engineer Mr. Randika Pathirana and all the staff of the Department of Computer Science and Engineering at the University of Moratuwa who helped me a lot in getting access to the university resources.

Last but not least, I would like to thank my parents and friends for their immense support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

DNA	Deoxyribonucleic Acid
NGS	Next Generation Sequencing
GPU	Graphics Processing Unit
I/O	Input/Output
SSD	Solid State Drive
DRAM	Dynamic Random Access Memory
FPGA	Field-Programmable Gate Array
NVM	Non-Volatile memory

1 INTRODUCTION

Genomics is the study of an organism's genome (complete set of DNA). Genomic research has made impressive strides in the medical field by facilitating researchers to study the role of genetic factors in complex diseases (e.g., cancer) and drug response. The availability of large-scale DNA sequence data and computational power to process such large-scale data are the primary factors that made genomics possible.

1.1 DNA and DNA Sequencing

A DNA molecule consists of two strands that form a double helix structure (see Figure 1.1). Each strand is composed of nucleotides and each nucleotide contains a sugar, a phosphate group, and a nitrogenous base. There are four different types of nitrogenous bases: adenine (A), cytosine (C), guanine (G), and thymine (T). The biological information required for an organism to develop, survive, and reproduce is stored in these bases. DNA sequencing is the process of determining the order of these bases which is crucial for genomic research.

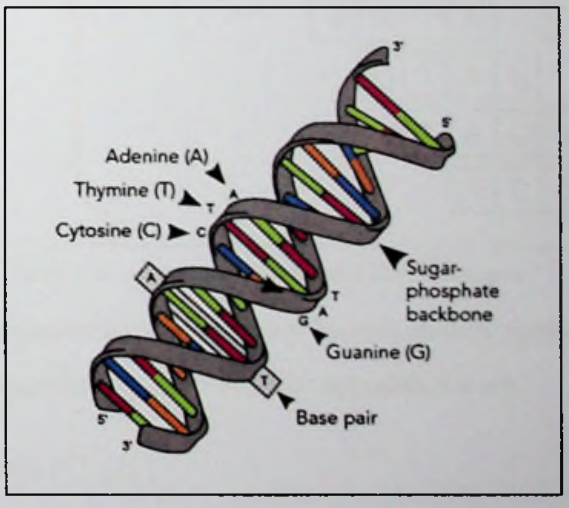

Figure 1.1: DNA double helix structure
Source: See Appendix II

1.2 *K*-mers and *K*-mer Counting

K-mers are substrings of length *K*. Given the DNA sequence *ACGAGGTACGA*, the following set of 4-mers can be generated: {*ACGA, CGAG, GAGG, AGGT, GGTA, GTAC, TACG, ACGA*} (see Figure 1.2). Suppose the length of the sequence is *L*, the total number of resulting *k*-mers is $(L - k + 1)$.

K-mer counting is the process of finding the frequency of each *k*-mer. Performing 4-mer counting on the DNA sequence *ACGAGGTACGA* would produce {*ACGA*: 2, *CGAG*: 1, *GAGG*: 1, *AGGT*: 1, *GGTA*: 1, *GTAC*: 1, *TACG*: 1}.

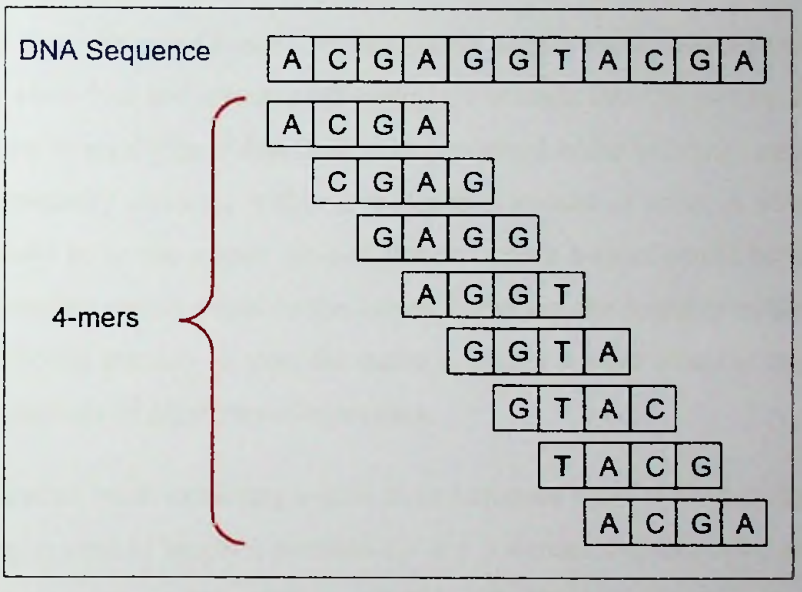

Figure 1.2: All possible 4-mers in a DNA sequence

K-mer counting is a fundamental step in many bioinformatics applications that process and analyze the DNA sequence data. Some of its applications are:

- Genome assembly [1], [2]
- Sequence error correction [3], [4]
- Repeat detection [5], [6]
- Multiple sequence alignment [7]
- Genome size prediction and estimation of genomic characteristics [8], [9]

1.3 The Problem

Human Genome Project [10] which was the first attempt to decipher the human genome was a 13-year long project (1990 - 2003) and cost $2.7 billion. However, with the development of massively parallel Next Generation Sequencing (NGS) technologies [11], the human genome can be sequenced within hours at the cost of a few thousand dollars. The high-throughput sequencing ability of NGS at a very low cost has resulted in the availability of large-scale genomic data which is continuously growing in size. Thus, it is inevitable for tools that process sequence data (such as *k*-mer counters) to be faster and more efficient.

Even though the problem of *k*-mer counting seems simple and straightforward, it is very challenging when time and resource efficiency are brought into the picture. Hundreds or even thousands of gigabytes of data need to be processed while utilizing limited hardware resources, especially memory, within a reasonable amount of time. A straightforward approach would be to use a hash table/dictionary where *k*-mers would be the keys and their corresponding counts would be the values. However, the majority of the systems do not have sufficient memory to store the entire counting data structure in memory when processing hundreds of gigabytes of input data.

Data amplification when extracting *k*-mers from sequence reads is another challenge in *k*-mer counting. A read of length L contains $L - k + 1$ *k*-mers. For example, assuming that one byte is used to store a character, a read of length 100 requires 100 bytes of memory space. However, the 20-mers extracted from that read, 81 in total, require 1620 bytes of memory space which is a 16.2x increase in volume.

A lot of research has been done in the past decade to optimize *k*-mer counting (e.g., Jellyfish [12], Gerbil [13], DSK [14], and KMC3 [15]). The majority of the existing *k*-mer counters focus on large *k* values ($k >= 20$) and optimize the algorithms to perform better for large *k* values [13]. Promising results (often for *k* values of 28 and 55) have been presented by many *k*-mer counters by adapting different optimization techniques.

However, such techniques often impose undesirable overhead when processing small k values ($k < 20$), leaving a gap that needs to be bridged.

1.4 Motivation

Since k-mer counting is one of the fundamental preprocessing steps in many bioinformatics applications, accelerating k-mer counting would benefit a wide range of applications. Even though k-mer counting for large k values is becoming popular, there are ample applications that rely on small k values [3], [9].

Except for a few k-mer counting tools (e.g., KMC2 and KMC3), the rest of the tools use a single algorithm for counting k-mers of all sizes. Instead of employing a single k-mer counting algorithm for all k values, using different yet suitable algorithms/optimization techniques for different ranges of k values would maximize the performance.

1.5 Objectives

The primary objectives of this research are listed below.

- Develop an efficient data representation and storage mechanism that minimizes I/O overhead for DNA sequence data.
- Design a cache-efficient parallel processing pipeline to load, count and store k-mers and their counts.
- Develop an open-source tool capable of counting and querying k-mers in a shared memory environment.
- Evaluate the correctness and performance of the developed tool using real-world datasets.

1.6 Contribution

Here, I present Frigate, a fast in-memory tool for counting and querying k-mers in a shared memory environment. Frigate is written in C and freely available at https://github.com/Gunavaran/frigate under MIT license. It is evident from the results

that Frigate has a comparable or better performance compared to state-of-the-art *k*-mer counters.

1.7 Thesis Outline

The rest of the document is organized as follows: Section 2 explores the relevant literature, Section 3 describes the methodology in detail, Section 4 presents and discusses the results and finally, Section 5 concludes the document.

2 LITERATURE REVIEW

Properties of the *k*-mer counting problem make it an ideal research topic for experts in different fields of computer science. The *k*-mer counting problem requires huge amounts of data to be processed in minimal time with the limited available resources, especially main memory. Hence, it demands combined efforts from a wide range of computer science fields such as parallel and distributed computing, data compression, data storage, and performance engineering. *K*-mer counting has gained popularity in the last decade and a significant number of algorithms/tools have been developed to efficiently count the *k*-mers. Efforts have been made to improve the performance through different approaches such as utilizing better data structures, data compression techniques to minimize the I/O overhead, efficient multithreaded processing pipelines, and incorporating modern hardware (e.g. GPUs).

Hashing and sorting are two popular approaches used in *k*-mer counting tools. Hashing-based tools store the *k*-mer counts in a hash table where *k*-mers are the keys and their corresponding counts are values. Sorting-based approaches sort the *k*-mers before counting them. It brings all the occurrences of the same *k*-mer to consecutive memory locations which results in fast counting. *K*-mer counting tools can also be categorized as disk-based or in-memory approaches. Disk-based approaches have become more popular and often outperform their in-memory counterparts. Unlike main memory, disk space is rarely considered a limited resource in modern computers. Disk-based approaches first partition the *k*-mers into multiple temporary files and then count the individual files independently. The *k*-mer counts of each temporary file are later merged if required.

2.1 In-memory Approaches

Jellyfish [12] is one of the earliest attempts to successfully accelerate *k*-mer counting. It employs multithreading, lock-free data structures (queues and a hash table), and data compression schemes to reduce the execution time and memory consumption. A multithreaded, lock-free hash table with open addressing is used to store the *k*-mers and their corresponding counts. To save memory, a bit-packed data structure is used to store

the *k*-mers instead of being aligned with computer words. Thread synchronization is achieved through compare and swap (CAS) instruction (which will be discussed later in Section 3). If the memory is not enough to count the whole dataset, intermediate results (hash tables) are written to the disk as files containing the (key, value) records. Once all the *k*-mers are counted, these intermediate files are merged. To facilitate the merging process and also to efficiently query the *k*-mer counting results, the hash table is sorted before being written to the disk. Jellyfish is often used as the benchmark to compare the performance of other *k*-mer counters.

CHTKC [16] is another hashing-based algorithm for *k*-mer counting. CHTKC uses a single reader thread to load blocks of sequence data from the disk and store them in a buffer. Multiple counter threads are used for counting the *k*-mers and finally, a single writer thread is used to store the counting results back in the disk. CHTKC outperforms Jellyfish primarily through better hash table design. It uses a lock-free hash table (similar to Jellyfish) and chaining is used to resolve collisions. Figure 2.1 demonstrates the hash table structure. A pre-allocated array of nodes is used to store the *k*-mers and their corresponding counts. A node also stores the index of the next linked node (to resolve collisions through chaining). The hash table array stores the indices of these nodes. In Figure 2.1, threads T1, T2, and T3 extract *k*-mers from the sequence data and the corresponding index in the hash table array is calculated using (1), a simple hash function.

$$index(M) = \left(\sum_{i=0}^{n} M_i\right) \bmod L \qquad (1)$$

where L is the size of the array in Figure 2.1, M is the *k*-mer and M_i is the i-th integer when M is encoded using n unsigned integers. Threads T1 and T2 are inserting *k*-mers that do not exist in the hash table yet. Hence, new nodes are requested from the pre-allocated node array and the indices of those nodes are stored in the hash table array. Thread T3 conflicts with thread T2 as both of them are mapped to the same index of 322. Hence, after thread T3 updates node 232, it is linked to node 150. Once the node array is full and new nodes

can no longer be obtained, the hash table is written to the disk. The remaining sequence data are stored in the disk in compressed form and are counted with the next new batch.

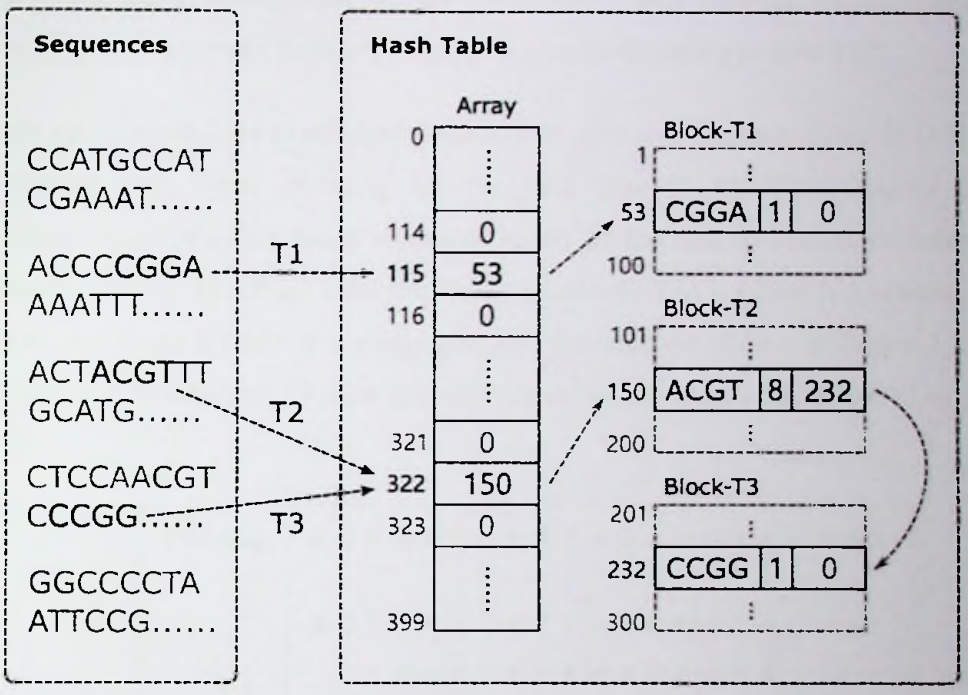

Figure 2.1: CHTKC hash table structure
Source: see Appendix II

2.2 Disk-based Approaches

Disk-based *k*-mer counters typically have two phases: (1) distributing the *k*-mers among multiple temporary files/partitions and (2) performing *k*-mer counting on each partition individually. When sequence reads are partitioned, the volume of the data amplifies substantially which results in significant I/O overhead. Hence, a lot of work has been done on efficiently distributing and storing the *k*-mers on the disk.

DSK [14] is a disk-based algorithm that utilizes hash tables for *k*-mer counting. To the best of my knowledge, DSK is the first successful attempt to partition the *k*-mers into temporary files and store them in the disk before processing (counting) each temporary file one after the other. DSK distributes the *k*-mers among multiple partitions in several iterations. The total number of iterations is decided by the total number of *k*-mers, the

amount of memory used to store a *k*-mer, and the allocated disk space. Similarly, the number of partitions is calculated using the total number of *k*-mers, memory required to store a *k*-mer and its count, total number of iterations, and allocated memory. DSK's performance was improved further by integrating multi-threading and an SSD.

The concept of *minimizers* to efficiently store DNA data was first introduced in [17] and was adopted into *k*-mer counting for the first time in MSPKmerCounter [18]. MSPKmerCounter is a disk-based algorithm based on the idea of Minimum Substring Partitioning (MSP). Minimum substring (aka minimizer) *t* of a *k*-mer is a substring of length *m* ($m < k$) such that *t* is lexicographically the smallest *m*-mer. In Figure 2.2, the DNA sequence is split into 17-mers and the minimum substring of all those 17-mers is *AACC*.

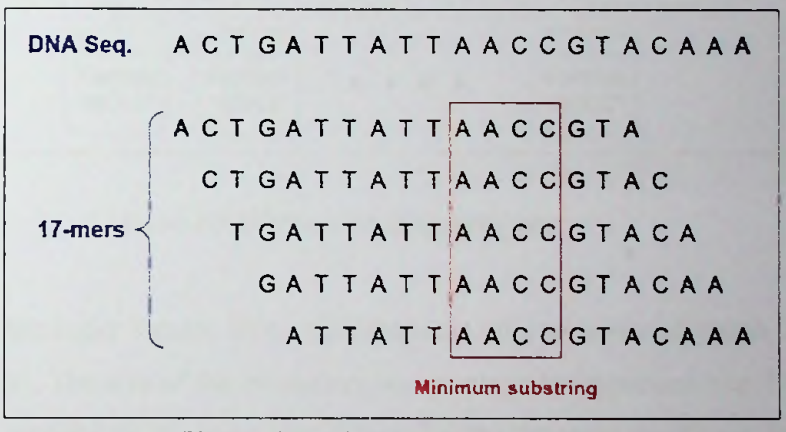

Figure 2.2: Minimum substring

Given a DNA sequence, MSP breaks it into super *k*-mers. A super *k*-mer is a substring of maximal length such that all the *k*-mers in the substring share the same minimizer. Consider the example in Figure 2.3. The DNA sequence is split into seven 16-mers. The first four 16-mers share the minimizer *ACAC* while the remaining three share the minimizer *ACCC*. Assuming that each character requires 1 byte of memory, it requires 112 bytes (16 * 7) to store all the 16-mers. Instead of splitting them into individual 16-mers, MSP splits them into two super *k*-mers (*CTGACACTTGACCCGTGGT*,

CACTTGACCCGTGGTCAT). To store these super *k*-mers, only 37 bytes are required, which is a significant reduction in the required amount of memory. This property substantially reduces the I/O cost.

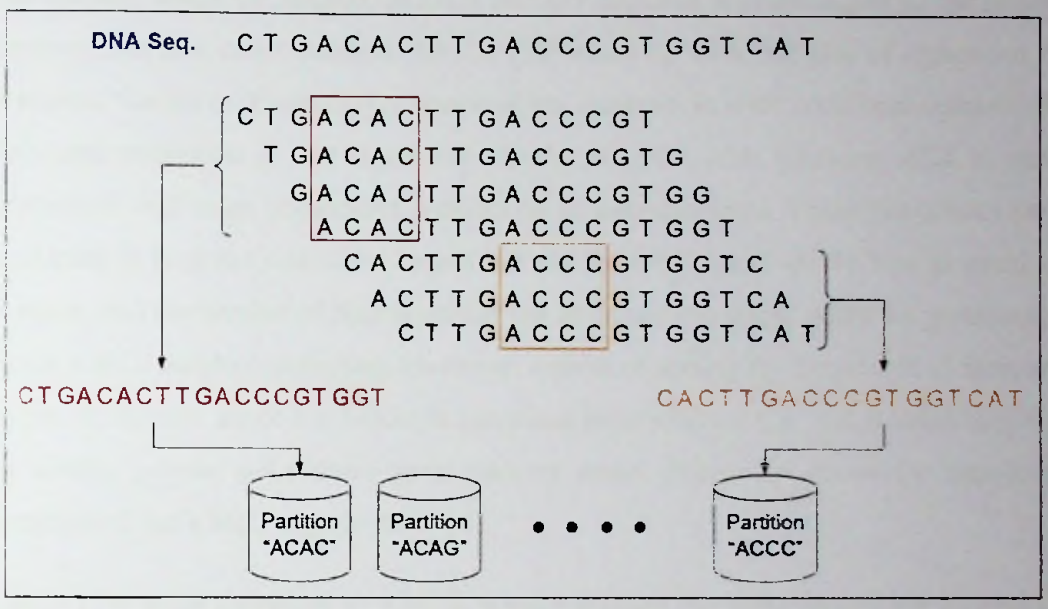

Figure 2.3: Minimum substring partitioning

After generating the super *k*-mers, those with the same minimizer are stored in the same partition (disk file). The size of the minimizer, say *p*, plays an important role. Smaller *p* will increase the probability of consecutive *k*-mers having the same minimizer. However, it might lead to an unbalanced distribution of *k*-mers and the largest partition might not fit into the main memory. Larger *p* will facilitate the distribution of partition sizes to be more even at the cost of reducing the probability of consecutive *k*-mers sharing the same minimizer. Hence, choosing the right size of the minimizer is crucial. After the partitioning phase, a hash table is used to perform *k*-mer counting where *k*-mers are the keys and their corresponding counts are the values. Another important property of this approach is that all the occurrences of the same *k*-mer are in the same partition. Hence the partitions can be counted individually and there is no need to merge the files.

The major drawback of MSPKmerCounter is that the distribution of partition sizes is far from uniform. Partitions associated with certain minimizers tend to be huge. For example, it was observed that the partition associated with the minimizer $AAA\ldots.A$ is usually larger than the rest. Since the amount of main memory required is proportional to the largest partition, this is a major concern. KMC2 [19] came up with the idea of *signatures* to overcome this issue. Essentially, signatures are minimizers with additional restrictions. One such restriction is that signatures should not start with AAA nor ACA as such minimizers lead to an unbalanced distribution of partition sizes. These restrictions help signatures to keep the size of the largest bin and the total size of all the bins as small as possible, and the number of bins is neither too large nor too small. After the partitioning phase, KMC2 employs radix sort. However, instead of sorting the k-mers, (k, x) mers are sorted. (k, x)-mers are $(k + x')$-mers in canonical form where $0 \leq x' \leq x$. It helps improve the sorting process and reduces peak memory usage. Figure 2.4 shows the high-level workflow of the KMC2 algorithm.

KMC3 [15] is the successor of KMC2. KMC3 follows the same processing scheme as KMC2, however, with improvements to the radix sort and k-mer partitioning.

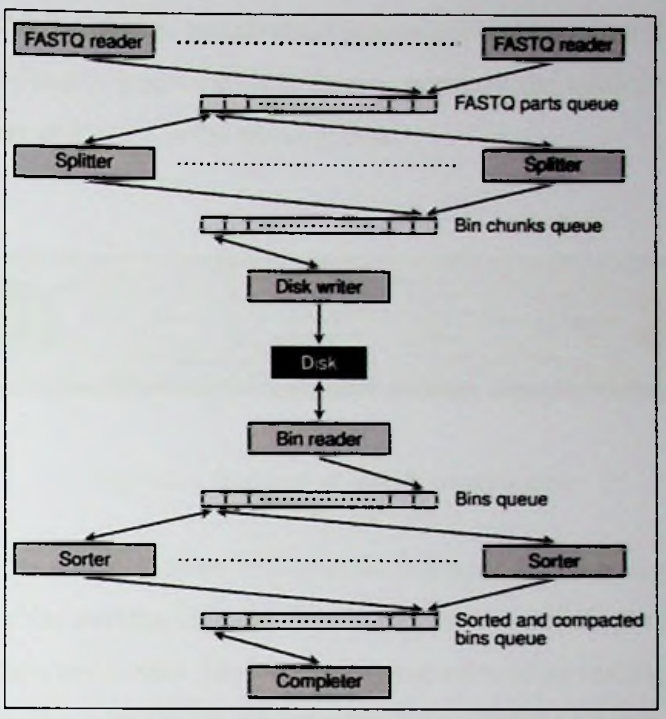

Figure 2.4: KMC2 parallel processing workflow
Source: See Appendix II

Gerbil [13] is another disk-based *k*-mer counter that uses hash tables for counting. Gerbil used two disks, one to store the input dataset and the final *k*-mer counts (input/output disk) and the other to store the temporary files created during the counting process (working disk). Similar to other disk-based counters, Gerbil has two phases: distribution and counting. In the distribution phase, minimizers are used to ensure that all occurrences of a *k*-mer are stored in the same temporary file. The workflow of the distribution phase is shown in Figure 2.5. A set of reader threads load the sequence data from the disk to the

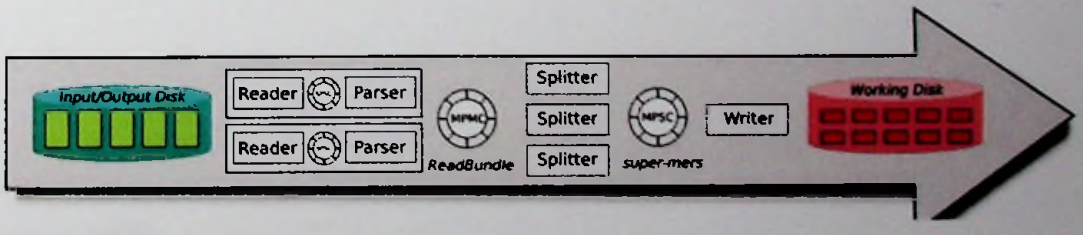

Figure 2.5: Workflow of Gerbil's distribution phase
Source: See Appendix II

main memory. Next, the data is transformed into an internal read bundle format by a set of parser threads. Finally, a set of splitter threads compute the minimizers, and a writer thread stores the super k-mers in the working disk.

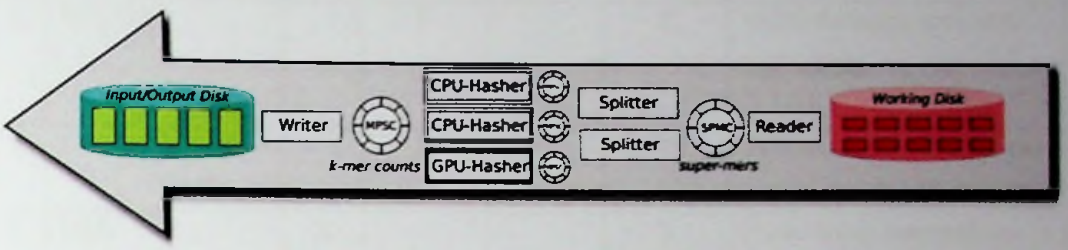

Figure 2.6: Workflow of Gerbil's counting phase
Source: See Appendix II

The workflow of the counting phase is illustrated in Figure 2.6. A reader thread loads the super k-mers from the working disk to the main memory. A set of splitter threads extract the k-mers from the super k-mers. Each k-mer is assigned to one of multiple hasher threads based on the hash value of the k-mer which ensures that all the occurrences of the same k-mer are assigned to the same hasher thread. Each hasher thread maintains a local hash table and updates the k-mer counts based on the k-mers received from the splitter threads. Once a temporary file has been processed completely, a writer thread writes the contents of all the local hash tables to the input/output disk. Gerbil also supports GPU integration which further reduces the running time.

Table 2.1 summarizes the different aspects of k-mer counters and their contribution.

Table 2.1: Summary of *k*-mer counting tools

	Tool/Algorithm	Disk	In-memory	Sorting	Hashing	Contribution
1.	Jellyfish		X		X	Multi-threaded, lock-free hash tables.
2.	CHTKC		X		X	Efficient hash table design.
3.	DSK	X			X	Disk-based, low memory counting.
4.	MSPKmerCounter	X			X	Minimum substring partitioning.
5.	KMC2	X		X		Signatures and (k, x)-mers.
6.	KMC3	X		X		Improved radix sort and signatures.
7.	Gerbil	X			X	Efficient processing pipeline and hash table design. GPU integration.

2.3 Approximate *K*-mer Counters

Unlike the above-mentioned tools which perform exact *k*-mer counting, another group of *k*-mer counters which approximate the *k*-mer frequencies has also been proposed. *K*-mer counters that perform exact counting need to process and store all the *k*-mers either in memory or disk. Consequently, the amount of memory required will grow linearly with the input data size, making exact *k*-mer counters inefficient for large datasets. Even though approximate *k*-mer counters tradeoff accuracy for less memory usage and execution time, the error percentage is not significant. ntCard [20], an approximate *k*-mer counter,

achieved orders of magnitude faster execution compared to DSK with less than a 1% error rate. Some of the approximate *k*-mer counters are KmerEstimate [21], KmerStream [22], KmerGenie [23] and Squeakr [24].

2.4 Distributed *K*-mer Counting

Distributed-memory solutions are becoming popular in the bioinformatics domain due to the large amount of data being produced by NGS technologies. Approaches based on Big Data frameworks such as Apache Hadoop and Apache Spark have been proposed as solutions to problems involving large-scale biological data [25]–[27]. Several distributed *k*-mer counters have also been proposed in recent years [28]–[30]. Shared-memory solutions suffer from the limitations in the number of processing units and the size of main memory, especially when processing large datasets. Distributed approaches have access to more resources in clusters or supercomputers. Jellyfish took about a day to count 22-mers for a 3TB dataset from the 1000 Genome Project [31]. However, Bloomfish, a distributed *k*-mer counter, managed to count 22-mers for a 24 TB dataset in just an hour.

2.5 *K*-mer Counters Utilizing Modern Hardware

Latest *k*-mer counters also make use of modern hardware such as graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and byte-addressable non-volatile memories (NVMs). GPUs are massively parallel systems capable of running thousands of threads simultaneously. GPU-based *k*-mer counters aim to hide the PCIe latency and improve the compute to global memory ratio. *K*-mer counters Gerbil and KMC2-GPU [32] have proved that better performance can be achieved by integrating GPUs. KMC2-GPU achieved over 4x speedup compared to its CPU counterpart. Compared to GPUs, FPGA-based accelerators are power efficient and highly customizable. Mcvicar et al. have presented the design and implementation of a high-performing FPGA-based *k*-mer counter in [33]. NVMs are emerging memory technologies that combine the non-volatility of disk drives and speed approaching that of DRAM [34]. They are being investigated to replace

the existing memory technologies especially in systems that process large amounts of data. In [35], authors have presented a *k*-mer counter that exploits both NVMs and GPUs.

Even though distributed environments and new hardware integration provide a better performance, they have not been widely adopted into bioinformatics research mainly because of availability constraints. This research focuses on developing a shared-memory solution for exact *k*-mer counting.

3 METHODOLOGY

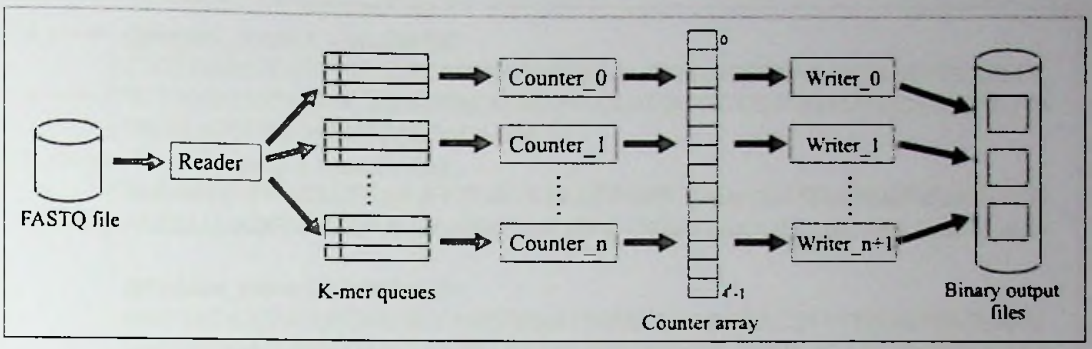

Figure 3.1: Parallel processing pipeline of Frigate

Frigate is a shared-memory tool that can perform *k*-mer counting sequentially or parallelly. The parallel processing pipeline of Frigate is illustrated in Figure 3.1. The three major steps in *k*-mer counting are reading the data from the disk, counting the frequency of *k*-mers, and writing the counting results back to the disk. Frigate uses a single reader thread and multiple counter and writer threads. The reader thread extracts the sequence reads from the FASTQ file (the FASTQ format is described later in this chapter) and distributes them uniformly among the *k*-mer queues. There is a one-to-one mapping between the *k*-mer queues and the counter threads. Counter threads extract the *k*-mers from the reads and update the counter array that is used to hold the *k*-mer frequencies. Once all the *k*-mers have been counted, both the counter threads and the reader thread act as writer threads to store the output of *k*-mer counting on the disk. The rest of this chapter is dedicated to explaining the design of each component and how they coordinate with each other to maximize performance.

3.1 FASTQ Files

FASTQ is a text format that is widely used to store biological sequence data. Each entry in a FASTQ file consists of four lines (see Figure 3.2):

1. A sequence identifier
2. The sequence or read
3. A separator ("+" sign)

4. The quality scores

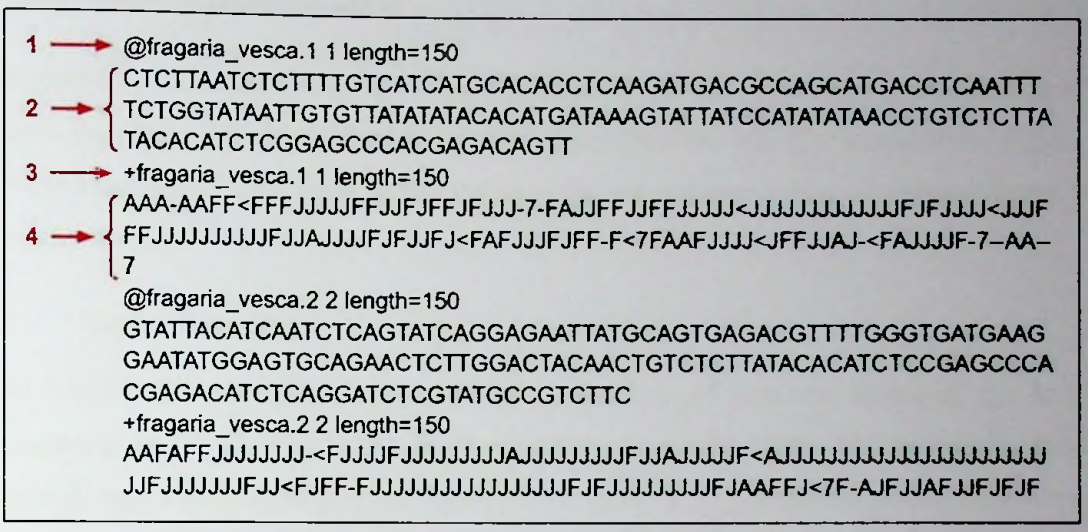

Figure 3.2: Two sample entries in a FASTQ file

In *k*-mer counting, we are interested in only the actual sequence (line-2). Hence, every fourth line starting from line-2 (assuming that line-1 is the first line) needs to be extracted from the FASTQ file by the reader thread.

3.2 Reader Thread and *K*-mer Queues

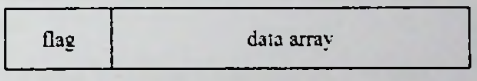

Figure 3.3: Structure of an entry in a *k*-mer queue

The *k*-mer queue is used as the medium of communication between the reader thread and the counter threads. A single entry in a *k*-mer queue consists of a flag and a data array of sufficient size to store the sequence reads as shown in Figure 3.3. Once the reader thread writes new data onto the data array, it would set the corresponding flag to notify the counter thread that new data is available to be processed. Once the counter thread has completed processing the sequence read in the data array, it would reset the flag to notify the reader thread that the data array can be replaced with new data. The default number of entries per *k*-mer queue is five, however, it can be modified as per requirement. This

design eliminates the need for synchronization between the reader and counter threads which results in faster execution.

Suppose all the *k*-mer queues are full and the reader thread has no free slots to write new data, the reader thread would act as a counter thread without idling. It would count *k*-mers from a number of reads equal to half the size of a *k*-mer queue before returning to being a reader thread.

3.3 Binary Encoding

In FASTQ files, each character is stored in one byte of memory. However, the four alphabets (A, C, G, and T) can be represented using only 2 bits. The counter threads encode each character as follows: {A:00, C:01: G:10: T:11}, while extracting the *k*-mers from the sequence reads in the *k*-mer queue. Figure 3.4 shows how the 12-mer *CTCTTAAGCTCT* is binary encoded.

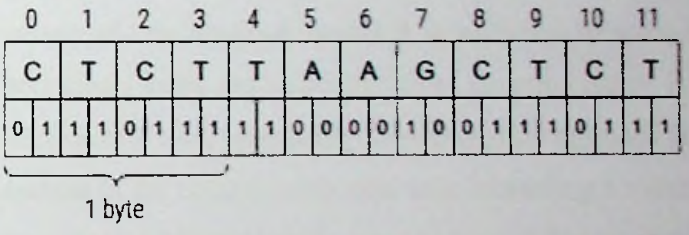

Figure 3.4: Binary encoding of a 12-mer

This encoding reduces the data volume fourfold. Further, binary encoded *k*-mers can be processed as integers rather than as strings. Processing integers is typically faster than processing strings.

3.4 Counter Array

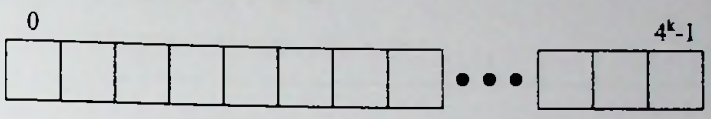

Figure 3.5: Structure of counter array

Frigate uses a linear array to store the *k*-mer counts as shown in Figure 3.5. There is a one-to-one mapping between every distinct *k*-mer and counter array index, i.e. every unique *k*-mer will be stored in a unique memory location. For a given *k*, the total number of possible *k*-mers is 4^k. Hence, the size of the linear array should be 4^k as well. Frigate stores the binary encoded *k*-mers in unsigned integers of sufficient size. The value of the unsigned integer decides the corresponding array index in which the *k*-mer will be stored. For example, the 6-mer *TTAGCA* will be encoded as 111100100100 and can be stored in a 16-bit unsigned integer. The decimal value of 111100100100 is 3876. Hence, the frequency of *TTAGCA* will be stored in the array index 3876.

The primary advantage of this approach is that it is sufficient to store only the *k*-mer counts. The corresponding *k*-mers can be derived from the array index. Furthermore, the counter array has O(1) access time complexity. However, this approach does suffer from the exponential increase in the counter array size with increasing *k* value. Since Frigate focuses on small *k* values ($k < 20$), modern systems do have sufficient memory to hold the linear array (assuming 1 or 2 bytes to store the counts).

3.5 Lock-free Counting

The traditional lock-based synchronization techniques often suffer from performance degradation due to convoying and priority inversion. Frigate uses Compare and Swap (CAS) instruction, a lock-free counting approach, to synchronize the multiple counter threads accessing the counter array. Thread synchronization using CAS primitives is typically faster than locking [36].

Algorithm 1: CAS

Inputs: *address*, *old*, *new*

```
    BEGIN ATOMIC
1       current ← value at address
2       if current = old
3               set address to new
4               return TRUE
5       else return FALSE

    END ATOMIC
```

CAS function takes three inputs: (1) memory location, (2) old value, and (3) new value, as shown in Algorithm 1. It reads the value at the given memory location and compares it with the second parameter (the old value). If both are equal, it would then replace the value in the memory location with the third parameter (the new value). If the memory location is successfully updated with the new value, it would return TRUE else return FALSE. All these steps are done atomically. Algorithm 2 shows how CAS operation is used to increment the counter array in Frigate.

Algorithm 2: INCREMENT

Inputs: *k_mer*

```
1       index = ENCODE(k_mer)
2       do
3           old_value = counter_array[index]
4       while !CAS (&counter_array[index], old_value, old_value + 1 )
```

3.6 Canonical *K*-mer Counting

Knowledge about the structure of DNA is necessary to understand what canonical *k*-mers are.

1. The DNA molecule has two strands where the base pairs are complements to each other. Nucleobase 'A' always pairs with 'T' and vice versa. Nucleobase 'C' always pairs with 'G' and vice versa.
2. When reading the nucleobases, the strands are read in opposite directions (from 5' end to 3' end) (see Figure 3.6).

For any observed sequence, its reverse complement exists in the opposite strand and either of them could have been sequenced. For example, in Figure 3.6, the left strand contains the 6-mer *TACAGT* and its reverse complement *ACTGTA* exists in the opposite strand. Encountering a sequence means that its reverse complement also exists in the data. Hence, instead of considering/storing a *k*-mer and its reverse complement separately, bioinformatics applications use the lexicographically smaller of the two known as the canonical *k*-mer. Frigate performs canonical *k*-mer counting by default (similar to other *k*-mer counters) and follows the lexicographic order A < C < G < T. Non-canonical *k*-mer counting can also be carried out by setting the relevant flag (-d) in the command line arguments.

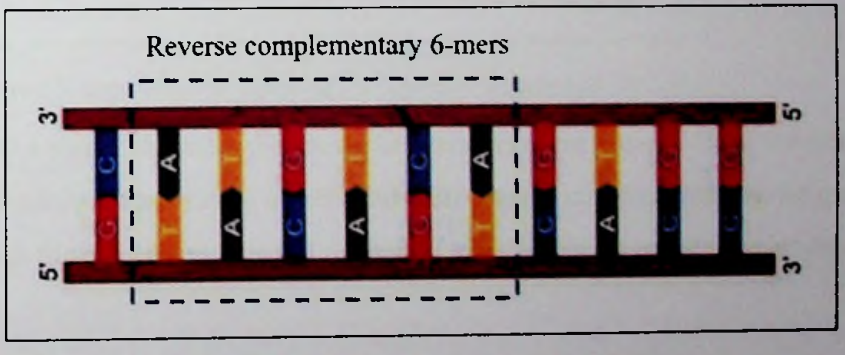

Figure 3.6: Structure of DNA with the complementary pairs
Source: See Appendix II

Finding the canonical *k*-mer of a given *k*-mer is one of the computationally intensive tasks in *k*-mer counting. Finding the canonical *k*-mer is a three-step process.

Step 1: Reverse the characters in the *k*-mer.

Step 2: Complement each character.

Step 3: Compare the original *k*-mer and its reverse complement.

Steps involved in finding the canonical sequence of the 6-mer *TACAGT* are presented in Figure 3.7. After generating the reverse complement *ACTGTA*, the *k*-mer and its reverse complement are compared. Since *ACTGTA* is lexicographically smaller, it is chosen as the canonical *k*-mer.

Frigate processes the *k*-mers in the same order as they appear in the read. The first *k*-mer of every read is binary encoded one character at a time and stored in an unsigned integer of sufficient size. While encoding each character, the reverse complement of the *k*-mer is also generated simultaneously and stored in a separate unsigned integer.

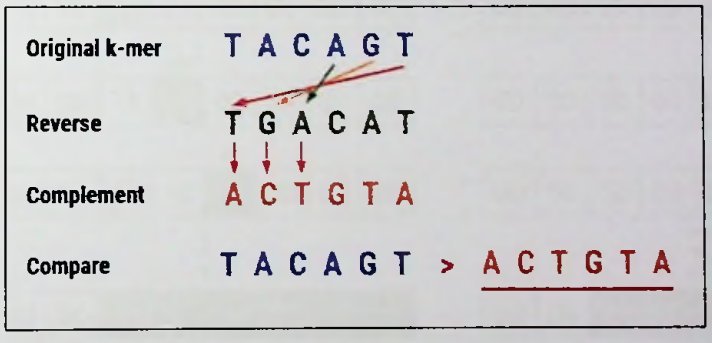

Figure 3.7: Steps involved in finding the canonical sequence of TACAGT

There is only a single character difference between adjacent *k*-mers. Once the first *k*-mer is binary encoded, the remaining *k*-mers and their reverse complements can be generated easily through bitwise operations. Let us look at an example for a better understanding.

Consider the read *TGAACGCAATGC* and let us assume that we are performing 6-mer counting. The first 6-mer is *TGAACG* and its reverse complement is *CGTTCA*. The steps involved in encoding both are illustrated in Figure 3.8. 12 bits are required to store a 6-mer. Hence two 16-bit unsigned integers are used to store the encoded *k*-mers. The corresponding bits are set using the operators 'bitwise OR' (|) and 'left shift' (<<). When encoding and storing 'T' in the variable, its complement 'A' is stored simultaneously in the variable that stores the reverse complement. All six characters are encoded similarly.

Once the first *k*-mer and its reverse complement are binary encoded, the remaining *k*-mers and their reverse complements can be generated through simple bitwise operations. Suppose that the first *k*-mer has been encoded, Frigate encodes the next *k*-mer in three steps.

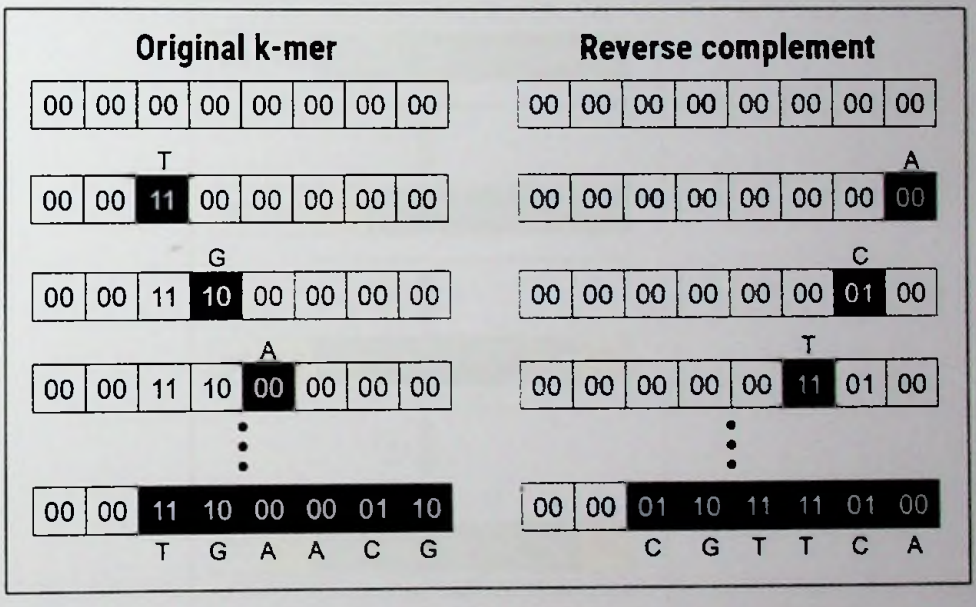

Figure 3.8: Encoding the first *k*-mer of a read and its reverse complement

Step 1: Reset the bits in the indices (2*k*-2) and (2*k*-1) from right to 0s using bitwise AND

Step 2: Left shift by 2 bits

Step 3: Set the last two bits based on the next alphabet using bitwise OR

Similarly, the reverse complement of the next *k*-mer is generated in two steps.

Step 1: Reset the bits in the indices (2*k*-2) and (2*k*-1) from right to 0s using bitwise AND

Step 2: Set the bits in the indices (2*k*-2) and (2*k*-1) from right based on the next alphabet using bitwise OR

The second *k*-mer of the read *TGAACGCAATGC* is *GAACGC*. Generating binary encoded *GAACGC* (second *k*-mer) from *TGAACG* (first *k*-mer) is shown in Figure 3.9. In step 1, the first two bits corresponding to 'T' are set to 00. In step 2, the integer is left shifted by two places to make room for the new character 'C' at the rightmost end. In step 3, the last two bits are set according to the new character to be appended, which is C.

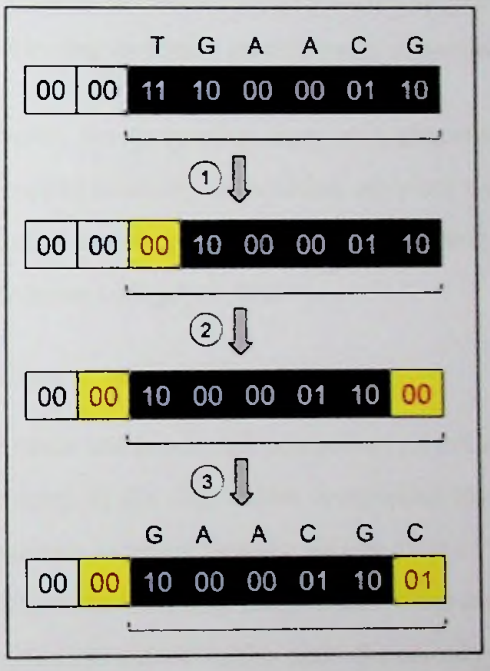

Figure 3.9: Encoding consecutive *k*-mers through bitwise operations

The workflow of generating the reverse complement of the second *k*-mer is shown in Figure 3.10. In step 1, the last two bits are removed by right shifting. In step 2, the first bits are set according to the new character to be added, which is G.

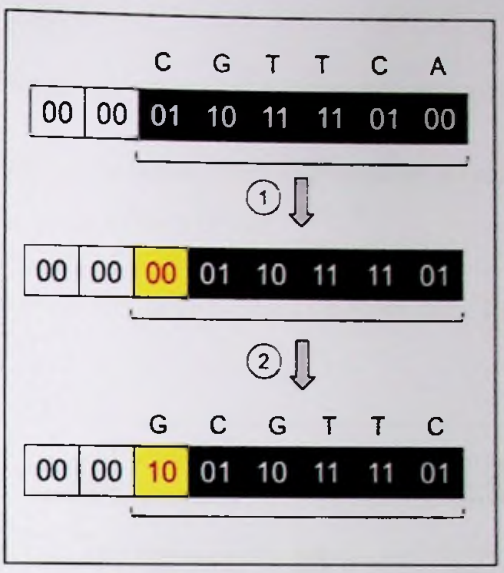

Figure 3.10: Encoding the reverse complements of consecutive *k*-mers through bitwise operations

Except for the first *k*-mer, the remaining ones in a sequence read and their reverse complements can be encoded similarly. Processing adjacent *k*-mers consecutively makes Frigate more cache efficient and also benefits from prefetching. Further, it eliminates the need to encode each character of every *k*-mer.

3.7 Storing *K*-mer Counts on the Disk

Once all the sequence reads are processed completely, a set of writer threads store the content of the counter array in the disk. After completing the counting phase, both the reader and counter threads act as writer threads. Unlike most of the *k*-mer counters, Frigate allows multiple threads to write the *k*-mer frequencies to the disk in separate files. Both *k*-mers and their frequencies are stored in the disk. Since only the *k*-mers with non-zero frequencies (or frequencies above a certain threshold) need to be stored, the counter array should be traversed to filter the non-zero frequencies. This can be achieved via two approaches,

Approach_1: Allocate an equal size, consecutive memory locations to each thread.

Approach_2: Allocate multiple, small segments of memory distributed throughout the array to each thread.

The division of work among four threads based on Approach_1 and Approach_2 is shown in Figure 3.11.

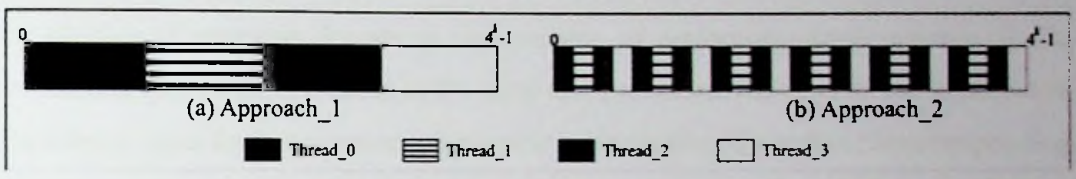

Figure 3.11: Two approaches for dividing the work among four writer threads.

The *k*-mers are not uniformly distributed in the counter array and often, the number of non-zero counts in each segment decreases as we move towards the end of the array. Since the number of non-zero counts is proportional to the number of disk writes, Approach_1 suffers from workload imbalance while Approach_2 does not.

To justify our claim, both approaches were implemented and tested on two real-world data sets *F. vesca* and *H. sapiens* with eight writer threads for k=15. Figure 3.12 shows the

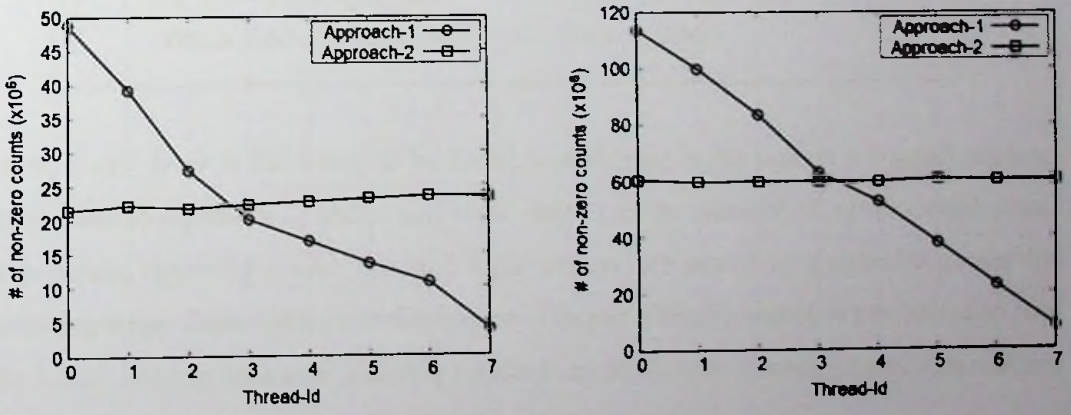

Figure 3.12: Number of non-zero counts encountered by eight writer threads after 15-mer counting for *F. vesca* (left) and *H. sapiens* (right) datasets.

number of non-zero counts encountered by each thread for both approaches when traversing the counter arrays after the counting phase. It is evident from the graphs that Approach_2 has a better workload balance than Approach_1. Hence, Frigate uses Approach_2 to store the *k*-mer counts in the disk.

All the occurrences of the same *k*-mer being stored in one file eliminates the need for merging the individual files generated by each writer thread.

3.8 Querying *K*-mers

In addition to counting, Frigate is also capable of performing *k*-mer queries. A file containing query *k*-mers is taken as the input and *k*-mers and their counts are displayed as the output. Since the *k*-mer counts are stored in multiple files, the output file corresponding to a *k*-mer must be identified before querying. As the writer threads follow a definite pattern when traversing the counter array, the correct output file can be identified through simple mathematical computations as shown in Algorithm 3.

Algorithm 3: QUERY

Inputs: *query_kmer, chunk_size, total_writers*

```
1   binary_kmer = ENCODE(query_kmer)
2   file_id = (binary_kmer / chunk_size) % total_writers
3   file_ptr = READ(file_id)
4   return BINARY-SEARCH(file_ptr, binary_kmer)
```

Here, *query_kmer* is the *k*-mer to be found, *chunk_size* is the size of the small segment mentioned in Approach_2 above and *total_writers* is the number of writer threads used. The details regarding *chunk_size* and *total_writers* are stored in a metafile during the counting stage. Once the correct file has been found, a binary search is performed to find the *k*-mer. Storing the *k*-mer counting results in multiple files of nearly equal size reduces the search space and facilitates the search.

3.9 Frigate Tool and its Functionalities

The implementation of Frigate along with its documentation is available at https://github.com/Gunavaran/frigate. Frigate is written in C and uses POSIX thread libraries for multi-threading. Some of the limitations of Frigate are:

- Only Linux based operating systems are supported
- Accepts only FASTQ format inputs
- The largest k value to be counted depends on the available main memory space

It provides four main functionalities: count, histogram, dump, and query.

1. **Count**

   The count functionality allows counting the frequency of k-mers. The command for count functionality is:

   ```
   ./frigate count [options] <input_file> <output_file>
   ```

   The list of available options is presented in Table 3.1.

Table 3.1: Available options for count functionality

Parameter	Description	Default
<input_file>	single FASTQ file	
<output_file>	preferred name for the output file(s)	
-k	k-mer length	10
-t	number of threads	# CPU cores
-m	max amount of RAM in GB	size of RAM
-l	exclude k-mers with count less than	2
-d	disable canonical counting	
-v	verbose mode; displays the parameter settings	false

2. **Histogram**

The histogram functionality computes (key, value) pairs where keys are frequencies and values are the number of distinct *k*-mers which have the specific frequency values. For example, (5, 4566) means 4566 distinct *k*-mers appear 5 times in the input data. A sample output of histogram functionality is shown in Figure 3.13. The command for histogram functionality is:

```
./frigate histogram <output_file> <histo_file>
```

The *k*-mer length need not be specified as it stored in the metafile while counting.

```
1 : 0
2 : 13794072
3 : 11254012
4 : 9655736
5 : 8169112
6 : 6845456
7 : 5744220
8 : 4863537
9 : 4157407
10 : 3579905
```

Figure 3.13: A sample output of histogram functionality

3. **Dump**

The dump functionality converts the output file to a readable ASCII format. A sample output of dump functionality is shown in Figure 3.14. The command for dump functionality is:

```
./frigate dump <output_file> <dump_file>
```

```
AAAAAAAAACCACAA    : 110
AAAAAAAAACCACAC    : 70
AAAAAAAAACCACAG    : 41
AAAAAAAAACCACAT    : 51
AAAAAAAAACCACCA    : 56
AAAAAAAAACCACCC    : 32
AAAAAAAAACCACCG    : 22
AAAAAAAAACCACCT    : 67
AAAAAAAAACCACGA    : 41
AAAAAAAAACCACGC    : 31
AAAAAAAAACCACGG    : 17
AAAAAAAAACCACGT    : 38
AAAAAAAAACCACTA    : 85
AAAAAAAAACCACTC    : 70
AAAAAAAAACCACTG    : 36
```

Figure 3.14: A sample output of dump command

4. **Query**

The query functionality takes a file containing a list of *k*-mers and returns the frequency of each *k*-mer. The command for query functionality is:

```
./frigate query <output_file> <query_file>
```

4 RESULTS

4.1 Experimental Setup

The performance of Frigate was compared with Jellyfish v2.3 [12], DSK v2.3.3 [14], Gerbil [13], KMC2 [19], KMC3 [15], and CHTKC [16]. The experiments were conducted on a machine with the configurations shown in Table 4.1. Each experiment was conducted three times and average running times were calculated. Suppose the execution time obtained from one iteration varied significantly from the remaining two iterations, it is omitted when calculating the average running time.

Table 4.1: Test machine configuration

Processor	Intel Xeon(R) E7-4820 v3 @ 1.90GHz
Threads per core	2
Cores per socket	10
No. of sockets	4
Main memory	64 GB
HDD	1.8 TB

To compare the performance of k-mer counters, canonical k-mer counting was performed on two real-world datasets *Fragaria vesca* (FV) and *Homo sapiens* (HS) (see Table 4.2). Additional information related to the datasets is presented in Appendix I.

Table 4.2: Datasets

Organism	Genome Size (Mb)	No. of bases (Gb)	FASTQ file size (GB)	Average read length (bases)	No. of reads
F. vesca (FV)	214	4.5	10.9	353	12,803,137
H. sapiens (HS)	2,991	123.7	292.1	151	819,148,264

Both small (FV) and large (HS) datasets were chosen to understand the impact of dataset size on the performance and behavior of *k*-mer counters. Note that, unlike the HS dataset, the entire FV dataset can be fit into the main memory.

The ambiguous nucleotides in DNA sequence datasets are denoted using the alphabet 'N' in FASTQ files. Similar to the existing *k*-mer counters, Frigate also ignores all the *k*-mers that contain these ambiguous nucleotides when counting.

Singleton *k*-mers (*k*-mers that appear only once in the dataset) are ignored when storing the counts on the disk. Singleton *k*-mers often result from sequencing errors [37]. It is very unlikely for a *k*-mer to appear only once when sequenced with high coverage. Further, these singleton *k*-mers are often uninformative for the majority of the bioinformatics applications that utilize *k*-mer counting. Since the percentage of singleton *k*-mers is high, storing their counts would impose a significant overhead. Out of the 255,498,918 distinct *k*-mers resulted from 15-mer counting for FV dataset, 75,473,967 (29.5%) were singleton *k*-mers. Similarly, out of the 585,126,337 distinct *k*-mers resulted from 15-mer counting for HS dataset, 313,172,922 (53.5%) were singleton *k*-mers. However, singleton *k*-mers can be included in the count by modifying the command line arguments of Frigate.

Since the counter array size is fixed for a particular *k* value, Frigate does not achieve any performance gain from having access to additional memory. Conversely, other *k*-mer counters benefit from it. Hence, for a fair comparison, all the *k*-mer counting tools were given access to the entire memory space available in the test machine. Since KMC3,

CHTKC, and DSK suffered from a large number of page faults, they were exposed to only 40GB of main memory.

4.2 Execution Time

Tables 4.3 - 4.8 present the running times of seven different *k*-mer counters using a different number of threads for two datasets FV and HS, and *k* values of 10, 15, and 17. These *k* values were selected to evaluate Frigate's performance for varying counter array sizes. Note that CHTKC and Gerbil require more than two threads, and sequential running times are not included for the HS dataset because of excessive time consumption. Execution times in bold indicate the best results for each thread count.

Table 4.3: *K*-mer counting execution time (in sec) for *F. vesca* ($k = 10$)

No. of threads	Jellyfish	DSK	Gerbil	CHTKC	KMC2	KMC3	Frigate
1	600	6,674	-	-	**74**	77	76
2	750	3,253	-	-	74	76	**49**
4	584	1,570	176	456	29	**27**	**27**
8	603	1,143	91	749	**12**	**12**	20
16	223	975	62	115	**9**	**9**	19
32	252	881	71	47	**10**	11	19

Table 4.4: *K*-mer counting execution time (in sec) for *H. sapiens* ($k = 10$)

No. of threads	Jellyfish	DSK	Gerbil	CHTKC	KMC2	KMC3	Frigate
2	15,614	88,214	-	-	**1,626**	1,627	2,059
4	17,493	44,032	5,627	9,923	**1,469**	1,571	1,524
8	16,197	28,855	5,060	16,756	**1,471**	1,577	1,488
16	10,997	22,461	5,471	2,513	**1,471**	1,630	1,482
32	8,084	19,389	5,129	1,708	**1,469**	1,621	1,474

Table 4.5: *K*-mer counting execution time (in sec) for *F. vesca* ($k = 15$)

No. of threads	Jellyfish	DSK	Gerbil	CHTKC	KMC2	KMC3	Frigate
1	1,599	6,510	-	-	531	617	**276**
2	965	3,508	-	-	383	442	**197**
4	546	1,695	174	973	183	203	**121**
8	277	948	79	937	102	115	**66**
16	148	573	51	179	68	74	**39**
32	83	496	44	80	52	59	**26**

Table 4.6: *K*-mer counting execution time (in sec) for *H. sapiens* ($k = 15$)

No. of threads	Jellyfish	DSK	Gerbil	CHTKC	KMC2	KMC3	Frigate
2	37,383	85,377	-	-	8,000	9,375	**4,800**
4	36,095	40,868	5,311	24,901	4,825	4,600	**3,071**
8	25,075	26,398	4,241	18,776	4,982	4,899	**1,830**
16	16,800	19,340	4,319	3,963	4,781	4,346	**1,610**
32	10,692	15,871	4,297	1,725	5,247	4,151	**1,561**

Table 4.7: *K*-mer counting execution time (in sec) for *F. vesca* ($k = 17$)

No. of threads	Jellyfish	DSK	Gerbil	CHTKC	KMC2	KMC3	Frigate
1	1,986	6,754	-	-	**559**	652	823
2	1,164	3,389	-	-	**392**	451	464
4	1,239	1,779	198	1,849	**187**	207	284
8	379	936	**81**	1,192	106	116	187
16	177	481	**46**	250	67	67	117
32	108	397	**43**	114	49	49	79

Table 4.8: *K*-mer counting execution time (in sec) for *H. sapiens* ($k = 17$)

No. of threads	Jellyfish	DSK	Gerbil	CHTKC	KMC2	KMC3	Frigate
2	59,792	74,916	-	-	8,192	9,726	**7,611**
4	27,330	40,215	7,572	94,028	4,516	**5,148**	6,842
8	11,505	24,556	4,702	23,475	3,903	4,149	**3,607**
16	6,115	16,602	4,130	5,475	3,890	3,912	**1,694**
32	4,077	13,246	4,095	2,283	4,057	3,583	**1,681**

Frigate has a comparable or better performance compared to the other *k*-mer counters. When the dataset is large (HS dataset), Frigate achieves up to 2-3x speedup compared to its competitors. KMC2, KMC3, and Gerbil perform slightly better for the FV dataset when $k = 17$. It occurs primarily because these *k*-mer counters suffer less I/O overhead because of small data size while Frigate deals with a large number of cache miss due to large counter array size. DSK is clearly the slowest. It was also observed that disk-based *k*-mer counters spend a larger portion of their running time on the distribution phase compared to the counting phase. For the HS dataset, KMC3 spent ~73% of the running time on the distribution phase when k=17.

The running times of different *k*-mer counters for the HS dataset are graphically illustrated in Figure 4.1 and Figure 4.2. The running times above 60,000s have been omitted from the plots for better visualization. It is evident from the graphs that *k*-mer counters that perform better consistently for all thread values (Gerbil, KMC2, KMC3, and Frigate) do not scale well beyond 8 cores for the larger dataset (note that when k=17, Frigate scales well up to 16 threads). When the number of counter threads is increased, the rate at which the reader thread loads data from the disk becomes a bottleneck. To overcome this limitation, counter threads were allowed to read directly from the disk. However, it led to degraded performance when executed with a large number of threads (16 or 32).

The execution times of CHTKC for the HS dataset are the closest to Frigate when 32 threads are used. However, when executed with 4 or 8 threads, CHTKC displays significantly higher running times compared to Frigate. Another interesting observation is the abrupt reduction in running times of CHTKC when the number of threads is increased from 8 to 16.

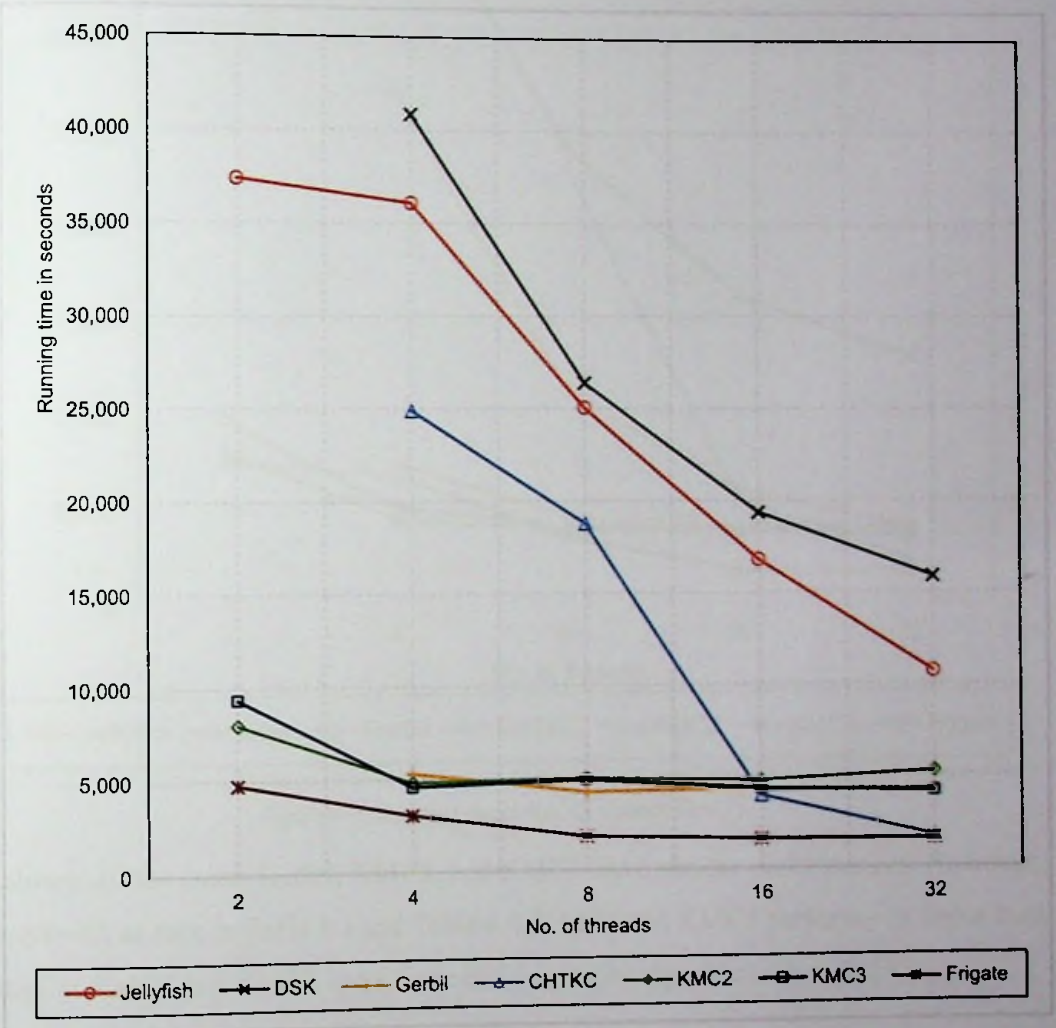

Figure 4.1: Running times for HS dataset (k=15)

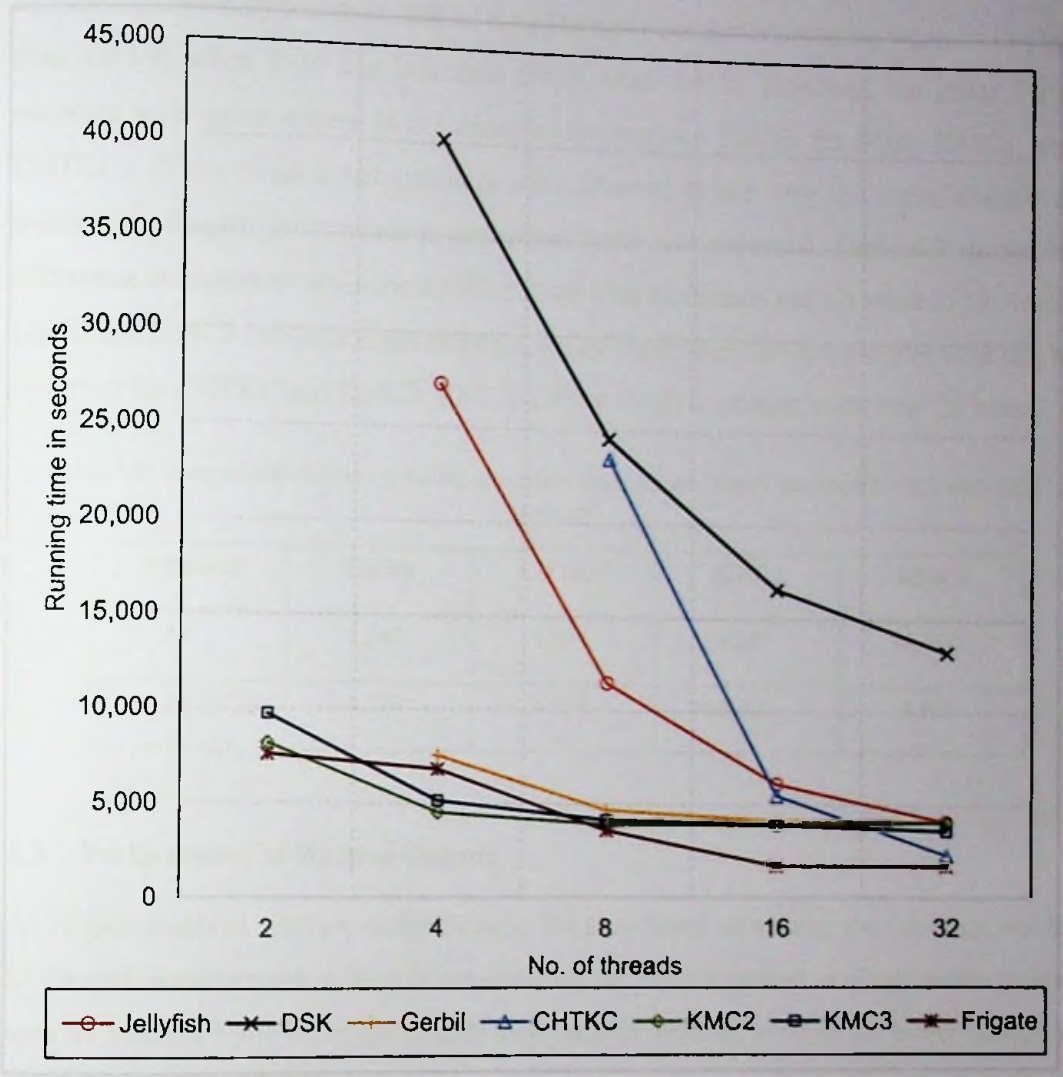

Figure 4.2: Running times for HS dataset (k=17)

In almost all the cases, Gerbil, KMC2, and KMC3 have similar performances. However, when k=10, as seen in Table 4.3 and Table 4.4, KMC2 and KMC3 perform ~3x faster than Gerbil as KMC2 and KMC3 use a separate in-memory algorithm when k<14.

Frigate requires only less than 2GB of RAM to carry out k-mer counting when k=15 (less than 10 MB when k=10 and less than 20GB when k=17). However, the other k-mer counters were given access to the entire memory space (40GB for DSK, KMC3, and CHTKC). When those k-mer counters were allowed to use only the same amount of memory as Frigate, an increase in execution times was observed. Table 4.9 shows the difference in execution times for the HS dataset with 32 threads and a k value of 15. While Gerbil and KMC3 exhibit a slight increase, a significant increase in execution time can be observed for CHTKC and KMC2. KMC2 did not finish execution even after 24 hours.

Table 4.9: Comparison of k-mer counting execution times (in sec) for *H. sapiens* (k = 15) with 2GB memory

# threads	Gerbil	CHTKC	KMC2	KMC3
32	4,297	1,725	5,247	4,151
32 (2 GB)	4,770	4,752	>86,400	4,916

4.3 Performance of Writing Output

As Frigate employs multiple writer threads, the time spent on writing the counting results to the disk is minimized. After 17-mer counting for the FV dataset, a single writer thread took 61 seconds while 32 writer threads took only 19 seconds to write the k-mer counts to the disk (see Figure 4.3).

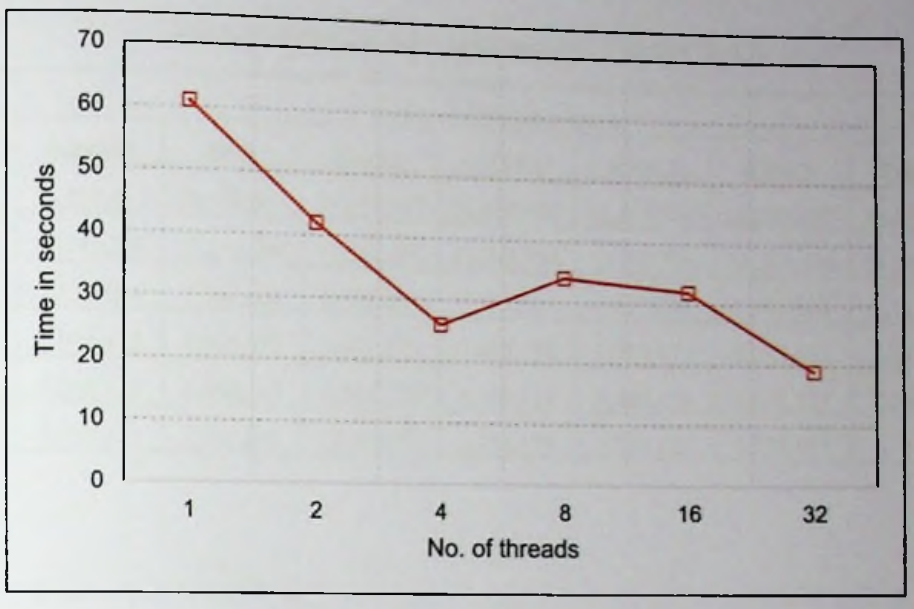

Figure 4.3: Time taken by different number of threads to write the k-mer counts to the disk after 17-mer counting of FV dataset

4.4 Result Validation

To evaluate the correctness, the k-mer frequency histogram (the total number of distinct k-mers with each given frequency) of Frigate was compared with those of Jellyfish, DSK, Gerbil, CHTKC, KMC2, and KMC3 for the FV dataset using the frequency range [2, 255] and k values of 15 and 17. Frigate exactly matched the results produced by the other k-mer counters. The k-mer frequency histogram for the FV dataset is provided in Table 4.10.

Table 4.10: *K*-mer frequency histogram for FV dataset (k=15)

Freq.	No. of distinct *k*-mers						
	Jellyfish	**DSK**	**Gerbil**	**CHTKC**	**KMC2**	**KMC3**	**Frigate**
2	28,038,487	28,038,487	28,038,487	28,038,487	28,038,487	28,038,487	28,038,487
3	13,243,796	13,243,796	13,243,796	13,243,796	13.243,796	13,243,796	13,243,796
4	7,532,004	7,532,004	7,532,004	7,532,004	7,532,004	7,532,004	7,532,004
5	5,085,473	5,085,473	5,085,473	5.085,473	5,085,473	5,085,473	5,085,473
6	4,068,118	4,068,118	4,068,118	4.068,118	4,068,118	4,068,118	4,068,118
7	3,758,651	3,758,651	3,758,651	3,758,651	3,758,651	3,758,651	3,758,651

5 CONCLUSION AND FUTURE WORK

K-mer counting is an essential component in many bioinformatics applications. Hence, a wide range of applications would benefit from accelerating *k*-mer counting. Even though the problem of *k*-mer counting is simple and straightforward from an algorithmic point of view, it becomes a challenging task when time and resource efficiency are brought into the picture. The need for processing hundreds or even thousands of gigabytes of data while utilizing limited resources such as main memory within a limited time makes *k*-mer counting an interesting research area for experts in the field of computer science.

K-mer counting has attracted a lot of interest in the past decade. *K*-mer counting algorithms that are optimized for large values of *k* impose additional overheads for small *k* values. In this thesis, I introduced Frigate, a tool for efficient counting and querying *k*-mers. Frigate is designed with an emphasis on *k* values less than 20 with a vision of using different algorithms for different ranges of *k* values to achieve optimal performance. Frigate is capable of both sequential and parallel processing. The processing pipeline of Frigate makes it cache efficient and the CAS technique is used to reduce the cost of thread synchronization.

The results clearly show that Frigate outperforms state-of-the-art *k*-mer counters in many instances achieving up to 2-3x times speedup, especially for large datasets, while consuming less memory space. It was also observed that the rate of data transfer between the disk and main memory becomes a bottleneck when a large number of threads are used for *k*-mer counting.

The first version of Frigate presented here only supports the Linux operating system and FASTQ file format. Support for other operating systems (e.g., Windows and macOS) and other file formats (e.g., FASTA) will be added in the future. Frigate will be further expanded by incorporating new algorithms for processing large *k* values. The performance of Frigate can be improved by incorporating data compression techniques and vectorization. Integrating GPUs is also a possibility to reduce the time consumed on

computing the canonical *k*-mers. The goal is to make Frigate an efficient toolset capable of performing a wide range of operations in the domain of bioinformatics.

REFERENCES

[1] R. Li *et al.*, "De novo assembly of human genomes with massively parallel short read sequencing," *Genome Research*, vol. 20, no. 2, pp. 265–272, Feb. 2010, doi: 10.1101/gr.097261.109.

[2] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "SWAP-Assembler: scalable and efficient genome assembly towards thousands of cores," *BMC Bioinformatics*, vol. 15, no. Suppl 9, p. S2, 2014, doi: 10.1186/1471-2105-15-S9-S2.

[3] D. R. Kelley, M. C. Schatz, and S. L. Salzberg, "Quake: quality-aware detection and correction of sequencing errors," *Genome Biology*, vol. 11, no. 11, p. R116, 2010, doi: 10.1186/gb-2010-11-11-r116.

[4] Y. Liu, J. Schröder, and B. Schmidt, "Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data," *Bioinformatics*, vol. 29, no. 3, pp. 308–315, Feb. 2013, doi: 10.1093/bioinformatics/bts690.

[5] S. Kurtz, A. Narechania, J. C. Stein, and D. Ware, "A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes," *BMC Genomics*, vol. 9, no. 1, p. 517, 2008, doi: 10.1186/1471-2164-9-517.

[6] A. L. Price, N. C. Jones, and P. A. Pevzner, "De novo identification of repeat families in large genomes," *Bioinformatics*, vol. 21 Suppl 1, pp. i351-358, Jun. 2005, doi: 10.1093/bioinformatics/bti1018.

[7] R. C. Edgar, "MUSCLE: multiple sequence alignment with high accuracy and high throughput," *Nucleic Acids Res*, vol. 32, no. 5, pp. 1792–1797, 2004, doi: 10.1093/nar/gkh340.

[8] M. Hozza, T. Vinař, and B. Brejová, "How Big is that Genome? Estimating Genome Size and Coverage from k-mer Abundance Spectra," in *String Processing and Information Retrieval*, vol. 9309, C. Iliopoulos, S. Puglisi, and E. Yilmaz, Eds. Cham: Springer International Publishing, 2015, pp. 199–209. doi: 10.1007/978-3-319-23826-5_20.

[9] B. Liu *et al.*, "Estimation of genomic characteristics by analyzing k-mer frequency in de novo genome projects," *arXiv:1308.2012 [q-bio]*, Feb. 2020, Accessed: Feb. 01, 2021. [Online]. Available: http://arxiv.org/abs/1308.2012

[10] F. S. Collins, "The Human Genome Project: Lessons from Large-Scale Biology," *Science*, vol. 300, no. 5617, pp. 286–290, Apr. 2003, doi: 10.1126/science.1084564.

[11] E. L. van Dijk, H. Auger, Y. Jaszczyszyn, and C. Thermes, "Ten years of next-generation sequencing technology," *Trends in Genetics*, vol. 30, no. 9, pp. 418–426, Sep. 2014, doi: 10.1016/j.tig.2014.07.001.

[12] G. Marçais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, vol. 27, no. 6, pp. 764–770, Mar. 2011, doi: 10.1093/bioinformatics/btr011.

[13] M. Erbert, S. Rechner, and M. Müller-Hannemann, "Gerbil: a fast and memory-efficient k-mer counter with GPU-support," *Algorithms Mol Biol*, vol. 12, no. 1, p. 9, Dec. 2017, doi: 10.1186/s13015-017-0097-9.

[14] G. Rizk, D. Lavenier, and R. Chikhi, "DSK: k-mer counting with very low memory usage," *Bioinformatics*, vol. 29, no. 5, pp. 652–653, Mar. 2013, doi: 10.1093/bioinformatics/btt020.

[15] M. Kokot, M. Długosz, and S. Deorowicz, "KMC 3: counting and manipulating k-mer statistics," *Bioinformatics*, vol. 33, no. 17, pp. 2759–2761, Sep. 2017, doi: 10.1093/bioinformatics/btx304.

[16] J. Wang, S. Chen, L. Dong, and G. Wang, "CHTKC: a robust and efficient k-mer counting algorithm based on a lock-free chaining hash table," *Briefings in Bioinformatics*, p. bbaa063, May 2020, doi: 10.1093/bib/bbaa063.

[17] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, Dec. 2004, doi: 10.1093/bioinformatics/bth408.

[18] Y. Li and XifengYan, "MSPKmerCounter: A Fast and Memory Efficient Approach for K-mer Counting," *arXiv:1505.06550 [cs, q-bio]*, May 2015, Accessed: Feb. 06, 2021. [Online]. Available: http://arxiv.org/abs/1505.06550

[19] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, "KMC 2: fast and resource-frugal k-mer counting," *Bioinformatics*, vol. 31, no. 10, pp. 1569–1576, May 2015, doi: 10.1093/bioinformatics/btv022.

[20] H. Mohamadi, H. Khan, and I. Birol, "ntCard: a streaming algorithm for cardinality estimation in genomics data," *Bioinformatics*, p. btw832, Jan. 2017, doi: 10.1093/bioinformatics/btw832.

[21] S. Behera, S. Gayen, J. S. Deogun, and N. V. Vinodchandran, "KmerEstimate: A Streaming Algorithm for Estimating k-mer Counts with Optimal Space Usage," in *Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, Washington DC USA, Aug. 2018, pp. 438–447. doi: 10.1145/3233547.3233587.

[22] P. Melsted and B. V. Halldórsson, "KmerStream: streaming algorithms for k -mer abundance estimation," *Bioinformatics*, vol. 30, no. 24, pp. 3541–3547, Dec. 2014, doi: 10.1093/bioinformatics/btu713.

[23] R. Chikhi and P. Medvedev, "Informed and automated k-mer size selection for genome assembly," *Bioinformatics*, vol. 30, no. 1, pp. 31–37, Jan. 2014, doi: 10.1093/bioinformatics/btt310.

[24] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "Squeakr: an exact and approximate k-mer counting system," *Bioinformatics*, vol. 34, no. 4, pp. 568–575, Feb. 2018, doi: 10.1093/bioinformatics/btx636.

[25] U. Ferraro Petrillo, G. Roscigno, G. Cattaneo, and R. Giancarlo, "FASTdoop: A Versatile and Efficient Library for the Input of FASTA and FASTQ Files for MapReduce Hadoop Bioinformatics Applications," *Bioinformatics*, p. btx010, Jan. 2017, doi: 10.1093/bioinformatics/btx010.

[26] G. Cattaneo, U. F. Petrillo, R. Giancarlo, and G. Roscigno, "An effective extension of the applicability of alignment-free biological sequence comparison algorithms with Hadoop," *J Supercomput*, vol. 73, no. 4, pp. 1467–1483, Apr. 2017, doi: 10.1007/s11227-016-1835-3.

[27] W. Zhou *et al.*, "MetaSpark: a spark-based distributed processing tool to recruit metagenomic reads to reference genomes," *Bioinformatics*, p. btw750, Jan. 2017, doi: 10.1093/bioinformatics/btw750.

[28] T. Gao *et al.*, "Bloomfish: A Highly Scalable Distributed K-mer Counting Framework," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, Shenzhen, Dec. 2017, pp. 170–179. doi: 10.1109/ICPADS.2017.00033.

[29] T. Pan, P. Flick, C. Jain, Y. Liu, and S. Aluru, "Kmerind: A Flexible Parallel Library for K-mer Indexing of Biological Sequences on Distributed Memory Systems," *IEEE/ACM Trans. Comput. Biol. and Bioinf.*, vol. 16, no. 4, pp. 1117–1131, Jul. 2019, doi: 10.1109/TCBB.2017.2760829.

[30] U. Ferraro Petrillo, M. Sorella, G. Cattaneo, R. Giancarlo, and S. E. Rombo, "Analyzing big datasets of genomic sequences: fast and scalable collection of k-mer statistics," *BMC Bioinformatics*, vol. 20, no. S4, p. 138, Apr. 2019, doi: 10.1186/s12859-019-2694-8.

[31] N. Siva, "1000 Genomes project," *Nat Biotechnol*, vol. 26, no. 3, pp. 256–256, Mar. 2008, doi: 10.1038/nbt0308-256b.

[32] H. Li, A. Ramachandran, and D. Chen, "GPU Acceleration of Advanced k-mer Counting for Computational Genomics," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Milan, Jul. 2018, pp. 1–4. doi: 10.1109/ASAP.2018.8445084.

[33] N. Mcvicar, C.-C. Lin, and S. Hauck, "K-Mer Counting Using Bloom Filters with an FPGA-Attached HMC," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, Apr. 2017, pp. 203–210. doi: 10.1109/FCCM.2017.23.

[34] J. Meena, S. Sze, U. Chand, and T.-Y. Tseng, "Overview of emerging nonvolatile memory technologies," *Nanoscale Res Lett*, vol. 9, no. 1, p. 526, 2014, doi: 10.1186/1556-276X-9-526.

[35] N. Cadenelli, J. Polo, and D. Carrera, "Accelerating K-mer Frequency Counting with GPU and Non-Volatile Memory," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Bangkok, Dec. 2017, pp. 434–441. doi: 10.1109/HPCC-SmartCity-DSS.2017.57.

[36] V. Gramoli, "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Francisco CA USA, Jan. 2015, pp. 1–10. doi: 10.1145/2688500.2688501.

[37] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in DNA sequences using a bloom filter," *BMC Bioinformatics*, vol. 12, no. 1, p. 333, Dec. 2011, doi: 10.1186/1471-2105-12-333.

APPENDICES

Appendix I: Datasets

F. vesca files were downloaded from the following URLs:

http://www.ebi.ac.uk/ena/data/view/SRX030576 (SRR072006, SRR072007)
http://www.ebi.ac.uk/ena/data/view/SRX030575 (SRR072005, SRR072010, SRR072011, SRR072012)
http://www.ebi.ac.uk/ena/data/view/SRX030577 (SRR072008, SRR072009)
http://www.ebi.ac.uk/ena/data/view/SRX030578 (SRR072013, SRR072014, SRR072029)

H. sapiens files were downloaded from the following URLs:

https://dnanexus-rnd.s3.amazonaws.com/NA12878-xten/reads/NA12878D_HiSeqX_R1.fastq.gz
https://dnanexus-rnd.s3.amazonaws.com/NA12878-xten/reads/NA12878D_HiSeqX_R2.fastq.gz

Appendix II: List of sources

Figure	Source
Figure 1.1	https://letstalkscience.ca/educational-resources/backgrounders/radiation-effects-on-cells-dna
Figure 2.1	[16]
Figure 2.4	[19]
Figure 2.5	[13]
Figure 2.6	[13]
Figure 3.6	https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.pinterest.com%2Fpin%2F536561743081402667%2F&psig=AOvVaw2k-QX8HtaTHyJdIcCIvZNF&ust=1620117116948000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCICY1dyMrfACFQAAAAAdAAAAABAD