International Journal of Computing and Digital Systems
ISSN (2210-142X)
Int. J. Com. Dig. Sys. 12, No.1 (Oct-2022)

https://dx.doi.org/10.12785/ijcds/120182

Performance Improvement of K-mer counting in DNA Sequence using Cache efficient Bl oom fil ter and recursive hash function

Elakkiya Prakasam[1] **and Arun Manoharan**[1]

[1]*School of Electronics Engineering,Vellore Institute of Technology, Vellore, Tamilnadu, India*

Received 22 Jan. 2021, Revised 15 Jul. 2022, Accepted 23 Jul. 2022, Published 31 Oct. 2022

Abstract:K-mer (k length substrings in a DNA sequence) counting plays an important role in genome assembly, sequence analysis, and error correction in sequence reads. In the Gene data sets, a single occurrence of k-mers occupies more storing space with a higher possibility of sequencing errors. Hence, error correction plays a significant role in eradicating such uninformative k-mers. Bloom filters data structure has been frequently used in k-mer counting for determining thek-mer occurence at least twice in a data set of a DNA sequence owing to its less memory usage and its fast querying. The standard bloom filer used in k-mer counting is not cache efficient as it accesses the whole bloom filter memory for single k-mer insertion/query. Also the Murmur hash consumes more time for hashing the k-mers from the Input Sequence. In this proposed work, we have improved the process of k-mer counting further by adopting different bloom architecture called a partitioned bloom data structure. The proposed architecture is cache efficient and uses only one memory access instead of in the standard bloom filte's k memory accesses. The rolling hash in ntHash function is used for hashing the k-mers from the input sequence has further reduced the hash computation time of k-mers. The proposed architecture was compared with standard architecture and the results showed that the proposed k-mer counter minimized significantly the k-mers loading and querying time from the memory for different data sets.

Keywords:K-mer counting, Bloom filter,Recursive Hash function,Genome Assembly.

1. Introduction

In Genomics, the data size of the genome has been increased tremendously. As the data size of the genome grows rapidly, it is challenging to store, distribute, and analyze the data using traditional computation methods. The data generated by the high throughput sequencing technologies doubles every year [1]. The gene sequences generated by the traditional Sanger sequencing [2] method is more time-consuming. Latterly, high throughput sequencing platform such as Illumina [3] has been developed to overcome the drawbacks of the traditional methods. This platform generates the gene data of a genome in the form of short reads. The short reads are assembled by various genome assembly approaches and it has been carried out with or without a reference genome. The Denovo assembly [4] of the genome is carried out without any reference genome. It has been carried out on Chrysanthemum eticusp using Illumina sequencing platform for genetic and gene discovery Analysis [5]. It produces exact sequences for even composite genomes. K-mers were generated from the short reads along with the frequency of occurrence. The k-mers are used to construct the Debrujin network, with the vertices being the k-mers and the edges connecting each overlapping k-mer. These de Bruijn graphs are used to give a theoretical groundwork that helps reduce the time required for computation. The only drawback with these graphs is that they are very large as they include a massive number of k-mers for genomes of vertebrates. The time taken for the entire process and the storage has become a major challenge in computation with large data sets. Many techniques have been proposed to overcome the problems in the De-novo genome assembly. In bioinformatics, especially for sequence analysis, sequence assembly, correction of errors in sequence reads, k-mer counting is considered as a cardinal element. K-mers are substrings (a small part of a large string) with length k. The number of k-mers generated in the l length short read sequence of is $l - k + 1$. In given n circumstances, the possible number of k-mers is nk. These k-mers are mainly used in the sequence alignment and sequence assembly. Many tools have been designed to count or estimate the occurrence of k-mers in a genome in less time with minimal hardware resources.

A lot of research is being carried out in bio-informatics using k-mer frequency counting. The implementation of k-mer counting has been categorized into two areas. One

E-mail address: elakkiyaece87@gmail.com, arunm@vit.ac.in

area is focusing on the shared memory and the hard disk in a single node (CPU) and the other one is focusing on the distributed environment. The shared memory architectures use multithreading in the CPU and the distributed architecture uses MPI to communicate between the nodes. Various k-mer counting tools developed in the past like Jellyfish[6], a k-mer counting tool used a lock-free hash table for storing the k-mer. The tool is implemented in multicore architecture to minimize the processing time by utilizing the compare and swap instructions in the system. Disk-based [7] tools have been developed to reduce the number of disk accesses at the expense of an increase in the I/O. The k-mer counting has been parallelized using splitter and reader threads. To further reduce the I/O and the computation time, KMC2 [8] is developed. This tool replaced the minimizer approach with the signature-based approach in which the disk space utility was minimized and data was stored in the SSD drives. This tool counted the k-mers with a k-mer length of 28. In this approach, multicores in the CPU were effectively utilized using OpenMP. It has been further accelerated by the hardware accelerators to reduce the computation time in the k-mer counting tool Gerbil [9]. This tool forms the histogram of k-mers ranging from 0 to 255 which were hashed and then stored in the hash table. The collisions in the hash table were handled by the double hashing technique. The parallel threads in the GPU were effectively utilized by using parallel and splitter thread models and also by distributing the work among the threads hence a high performance has been achieved. Also, the processing time for k-mer counting has been optimized using a trie data structure [10]. This trie structure used a signature-based approach for storing and retrieving the k-mers but the drawback of this approach was the increase in the storage for the trie data structure. The k-mer generation and counting have been further accelerated by implementation in the FPGA hardware accelerator [11]. The counting process is parallelized by utilizing k-cores in FPGA. An IP-Core has been generated for the k-mer generation and counting and implemented on FPGA. The availability of parallel comparators in FPGA has been utilized for counting which speeds up the performance. Several k-mer counting tools were reviewed and evaluated in terms of their relative performance,mainly on runtime and storage utility [12]. The review also considered additional criteria such as parallelism, disk usage, performance in terms of counting the frequency for higher k values, and their scalability to huge datasets. All the k-mer counting applications developed in the literature have targeted the optimization of storage requirement, time taken for insertion, and querying of the k-mers.

The bloom filter based k-mer counting tool, BFCounter [13], is implemented in the literature for counting k-mers with reduced memory utilization. The k-mers generated from the short reads are and stored in the bloom filter. The count of k-mers greater than one will be stored in the hash table. The Murmur hash function, a non-cryptographic hash function is normally used to hash the k-mers because of its less collision time, fast insertion, and querying speed. However, the bloom filter used in the literature has the drawback of poor cache locality due to its requirement of accessing h locations in the memory to query the existence of single k-mer. The murmur hash function has been used to hash the k-mers and it will be called each time when a k-mer from the short read is hashed. There is still more scope available for the improvement of the existing murmur hash functions in the standard bloom filter.

In our work, we have implemented a hybrid approach to enhance the process of k-mer counting. The proposed hybrid method used a partitioned bloom filter with a better cache locality. It uses only one memory access instead of h memory accesses. The ntHash function, a hash function for streaming k-mers, used in the hybrid method over murmur hash has significantly reduced the hash operation time.

The remaining article is organized as follows: The k-mer counting technique, bloom filter, hash function, and proposed methodologies are discussed in Section 2; Section 3 reports the results and discussion, and Section 4 draws the conclusion.

2. Methodology

The process of k-mer counting starts with the extraction of data from the fastq file. A fastq file is a text based file format which encompasses data sequences along with a quality score for every individual base, which is expressed as an ASCII character. The score of quality stretches from 2 to 40 which is an integer Q. In the fastq files, each single entry has four lines. The first line is the Sequence identifier; the read sequence comes in the second line, the third line has the quality score identifier and finally the quality score. The quality runs are depicted by bytes which are from 0x21 to 0x7e i.e., from the lowest quality to the highest quality. Fastq files are used to store the sequence reads extracted from the single strand of DNA. The k-mers are extracted from these reads by grouping sets of nucleotides depending on the value of k. Figure 1 depicts how the sequence read is grouped into k-mers. Here the value of k is 3.

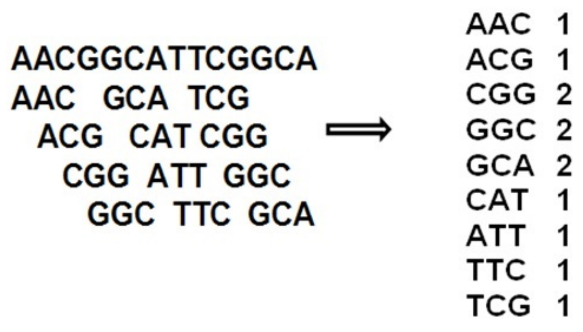

Figure 1. K-mer Counting with k=3

Now each k-mer is taken through a set of operations as mentioned in the algorithm and finally produces the k-mers along with the count.

A. K-mer counting

The K-mer frequency counting [13] utilizes a bloom filter based approach. The reads generated from the next generation sequencing technologies contain more errors due to the change in the single base (e.g. A is changed to T) during sequencing process. The error k-mer occurs only once in the sequence. The k-mers with single frequency contribute more to the frequency counting results. Thus the single occurrence k-mers need not be counted during k-mer counting. There are many error correction tools available for correcting the errors in the reads [14]. For example, tools mentioned by [15] and [16] incorporates bloom filter for error corrections. The process of counting k-mers usually involves two steps. The k-mers were extracted from the short reads and the canonical representations of k-mers were hashed and saved in the bloom filter. The k-mers occurring more than once is checked in the bloom filter; if it exists then it will be stored in the hash table. Thus, the single occurence k-mers will not be stored in the hash table resulting in a reduction in the storage space. Since the bloom filter is susceptible to false positive, a second pass is done to remove the false k-mers. The k-mers that occur only once were removed fromthe hash table at the end. The algorithm for k-mer counting is shown below.

Algorithm 1 K-mer Counter with Bloom filter

```
1: $B \leftarrow Bloomfilter$
2: $HT \leftarrow HashTable$
3: for all reads s do
4:     for all kmers $x$ in s do
5:         $xrep \leftarrow min(x, revcom(x))$
6:         if xrep $\epsilon$ B then
7:             if xrep $\notin$ HT then $T[xrep] \leftarrow 0$
8:             end if
9:         else
10:            add xrep to B
11:        end if
12:    end for
13: end for
14: for all reads s do
15:     for all kmers $x$ in s do
16:         $xrep \leftarrow min(x, revcom(x))$
17:         if xrep $\epsilon$ HT then
18:             $HT[xrep] \leftarrow HT[xrep] + 1$
19:         end if
20:     end for
21: end for
22: for all $x$ $\epsilon$ HT do
23:     if HT[x] =1 then
24:         remove x from HT
25:     end if
26: end for
```

B. Existing Methodology

In the existing k-mer counting method, for fast insertion and querying, standard bloom filter is used along with murmur hash function. The hashing process and the bloom filter is described in the following section.

1) Hashing

Hashing is a technique to generate random values using hash function which will be used for indexing the data into the specific data structure. It converts the string or a substring of characters that are extracted from the sequence reads of a DNA into a value which is the location in the data structure. This hashing methodology is used to alphabetize and bring back items in a database as it is quicker to search for an element that uses this hashed key. This hashing mechanism is used in many encryption algorithms.

Algorithm 2 Murmur3 Hash

```
1: Murmur3(key,len,seed)
2: c1=0xcc9e2d51;
3: c2=0x1b873593;
4: r1=15;
5: r2=13;
6: m=5;
7: h=0xe6546b64;
8: hash=seed;
9: for each fourBytechunkofkey do
10:     k=four Byte Chunk;
11:     k=k*c1;
12:     k =k rol r1;
13:     k=k*c2;
14:     hash=hash ⊕ k;
15:     hash =hash rol r2;
16:     hash=(hash*m)+n;
17: end for
18: for any remaining Bytes in Key do
19:     remaining Bytes= remaining Bytes * c1;
20:     remaining Bytes= remaining Bytes rol c1;
21:     remaining Bytes= remaining Bytes * c2;
22:     hash =hash ⊕ remaining Bytes;
23: end for
24: hash=hash ⊕ len;
25: hash=hash ⊕ (h>>16);
26: hash=hash ⊕ (h>>13);
27: hash=hash * 0xc2b2ae35;
28: hash=hash ⊕ (h>>16);
29: return hash
```

They are also massively used for querying and indexing in the fields of bioinformatics, k-mer counting, and assembly of transcriptome and correction of errors.There are various hashing mechanisms widely used in different applications but not limited to cryptographic applications, network applications etc., The murmurHash3, a non-cryptographic hashing function, is the best suited for simple hashing algorithm. The murmur hash function is shown in Algortihm 2. The murmur3 hash[17] is advantageous of being simple when considering the amount of generated instructions in the assembly. It has also good avalanche behavior and is resistant to collisions. The working of this hash is based multiplication and rotation operations. First, we take the k-mers with a particular value of k from the sequence reads in the DNA. The k-mers are hashed using murmur hash and stored in the bloom filter at the location given by the hash function.

2) *Bloom Filter Architecture*

Bloom filter is a probabilistic data structure which supports dynamic membership queries with less memory consumption allowing a certain amount of false positive occurrences. The bloom filter is used to reduce the memory utilisation by creating a trade-off between the memory and the false positive probability. A bloom filter [18] stores a set A of xi input elements of size n i.e., $A = xi|i = 1, 2, 3 \ldots n$ in an m bit array initialised to 0. Bloom filter maps each element from the set A into a random value with the range $1, 2, 3, 4.., m$ using K hash functions $h1, h2, \ldots, hk$. For all the elements in the set, the bit locations at $hi(x)$ are set to 1 for $1ik$. To check the existence of y in the set A, all the elements in the location $hi(y)$ are checked for the value of 1. If all the locations contain the value 1 then y is the element of A else it is not in the set as shown in Figure 2. For Example, the set A has elements abc, def and inserted into the bloom filter. While querying for the element 'pqr', the query result is not in the set since all the bits are zero but element 'xyz' is false positive as it is not in the set but mistakenly understood as element of the set.

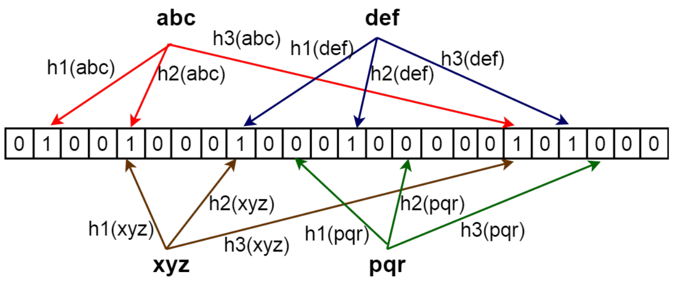

Figure 2. Bloom Filter Data Structure

The bloom filter will yield the result of the query with the false positive probability of f. The false positive probability [19] is expressed in equation1.

$$f = (1 - e^{-nk/m})^k \tag{1}$$

There exists a trade-off between the false positive probability and the bloom array (m) for a given number of inputs (n). The hash function (k) has been chosen based on 'm' and 'n' which can be derived as $k = (m/n) \cdot ln2$. The false positive probability can be approximated as $f \approx [(0.6185)]^{n/m}$. The novel applications of bloom filter and its design innovations were discussed in [20] .

C. *Proposed Methodology*

The bloom filter used in the k-mer counting method has poor cache locality and the hash function uses more computation time. To further improve the computation time of the k-mer counting process, the partitioned bloom filter and the nucleotide hash function is used and they are described in the following section.

1) *Nucleotide Hash (NtHash)*

NtHash [21] has exclusively been designed for the nucleotides. It is a cyclic polynomial hash function used for processing the sequences of DNA and RNA in bioinformatics applications. The ntHash performs in the best way while calculating the hash values of k-mers that are adjacent to each other in a sequence of a particular DNA. This function can periodically compute and obtain hash values of all the k-mers present in the sequence reads and provides large improvements in the performance during the hashing process.

Algorithm 3 ntHash

```
1: Nthash firstkmer (kmerSeq, k)
2: hash=0;
3: for i=0 to k-1 do
4:     hash = rol(h[kmerSeq(i)],k-1-i);
5: end for
6: return hash
7:
8: Nthash sliding(hash,charOut,charIn,k)
9: return rol (hash,1) ⊕ rol(h[charOut],k) ⊕ h[charIn];
```

The ntHash hash uses the recursive function f for hashing k-mers in a sequence r with length denoted by l. The hash function uses cyclic polynomial function for hashing operation and it uses barrel shifting rather than multiplication operations so as to make the process much quicker. The hash function for the computation of the hash values of the given k-mers in the sequence s of length l has been expressed in equation2 and 3.

$$H(k-mer_0) = rol^{k-1} \cdot h(r[0]) \oplus rol^{k-2} \cdot h(r[1]) \cdots \oplus h(r[k-1]) \tag{2}$$

The hash value of the previous k-mer is used to generate the hash value of the present k-mer:

$$H(k-mer_i) = rol^1 \cdot H(k-mer_{i-1}) \oplus rol^k \cdot h(r[i-1]) \oplus h(r[i+k-1]) \tag{3}$$

the term rol stands for cyclic binary left rotation, $h(.)$ is the seed table, and $\oplus$ stands for bit-wise exclusive OR operator. In this paper, NtHash algorithm was executed over murmur 3 hash because ntHash has less time complexity. It has $O(k + l)$ time complexity whereas the traditional hash functions have the time complexity $O(k.l)$ where k is the k-mer length and l is the read length of the sequence. The ntHash algorithm is shown in Algorithm 3.

2) *Partitioned Bloom filter*

Partitioned Bloom filter (Bloom-1 filter) is data structure [22] which performs membership check in a single memory access compared to 'h' memory accesses in a standard bloom filter. The primary ideology of this filter is that the bloom filter memory is divided into 'l' blocks. The block size is chosen based on the cache line size of 64 bytes. Each element from the set is hashed and updated in the 'h' memory locations in the single block instead of mapping randomly in the entire bit array. This bloom-1 filter is an array B, which has l number of words, with each word having the length of w bits. The Partitioned bloom filter is shown in Figure 3 .

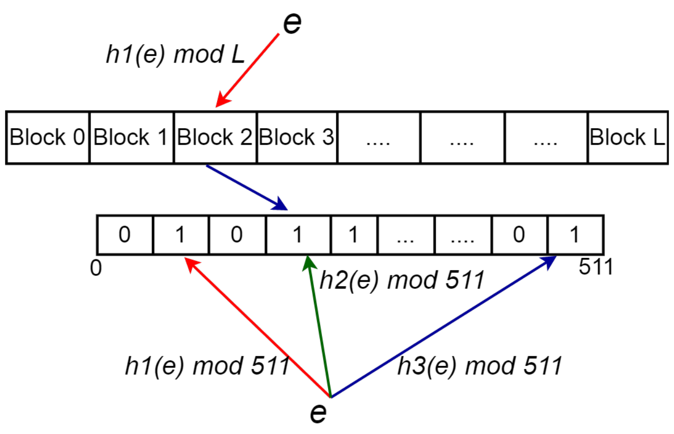

Figure 3. Partitioned Bloom Filter Data Structure

The total number of bits in the bloom filter can be written as lxw. For inserting an element into the filter, a membership block is selected from the hash value and then the element is hashed again h times and updates the bits in the block. For checking if an element is present in the bloom filter i.e., membership of the set, we perform hashing operation on the element. It requires $log_2 l + klog_2 w$ hash bits. In the bloom-1 filter, we use $log_2 l$ number of bits to locate the membership word and then $klog_2 w$ number of bits is required to do the membership check inside the word.If all the membership bits are ones, then it is a member of the setelse it is not a member.The false negative value of this algorithm is equal to zero. Hence with the usage of bloom-1 filter we can reduce the time complexity but with the added cost of increased false positive rate. The false positive probability [23] of the Partitioned bloom filter is given as

$$f(W,b,k) = \sum_{i=0}^{\infty} Poisson_{W/b}(i) \cdot f_{inner}(W,i,k) \qquad (4)$$

Where b denotes bits per element which is equal to m/n,h represents the number of hash function and W is the block size which is the cache line size. Figure 4 shows the number of memory access required for bloom filter which is linear but constant for bloom-1 filter.

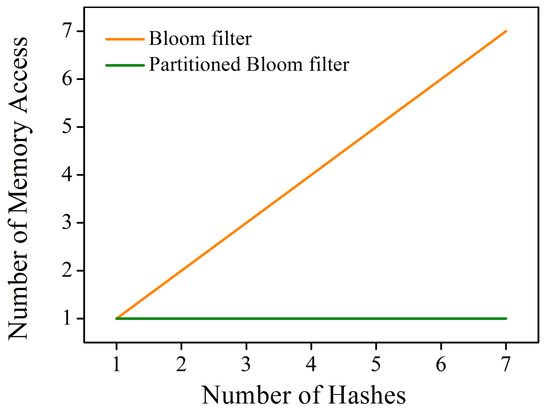

Figure 4. Memory Access for Bloom and Partitioned Bloom filter

3. Results and discussion

In this work, we have used FragariaVesca species gene sequence which is the scientific name of a wild strawberry plant for the performance analysis of k-mer counting. The input files shown in Table I were extracted from the database '(ftp://ftp.ddbj.nig.ac.jp/ddbj_database)'. The input files are Fastq formatted file.

TABLE I. Datasets used for K-mer counting process in DNA Sequence

Input Datasets	Number of Reads	Average read length	Size(GB)
Dataset 1	195472	367.551	0.478
Dataset 2	623466	338.802	0.986
Dataset 3	2285726	350.243	1.8
Dataset 4	4195487	327.748	3.1
Dataset 5	6321741	368.881	5.2
Dataset 6	12802954	352.074	10.2

This work is carried out on an Intel(R) Xeon(R) CPU(E5-2620v4, 2.10GHz with 8cores) with 32GB RAM and 1TB SDD running on Ubuntu 16.04 LTS machine. Since the choice of hash function decides the performance of bloom filter, we compared the murmur hash and Nthash functions as a function of k-mer processing time for different loads. A number of unique k-mer files of length k=27 were generated and given as input to the hash function. As seen from Figure 5, ntHash took very less processing time compared to murmur3 hash. From the findings, Nthash function is best suitable and specially designed for hashing consecutive strings like k-mers.

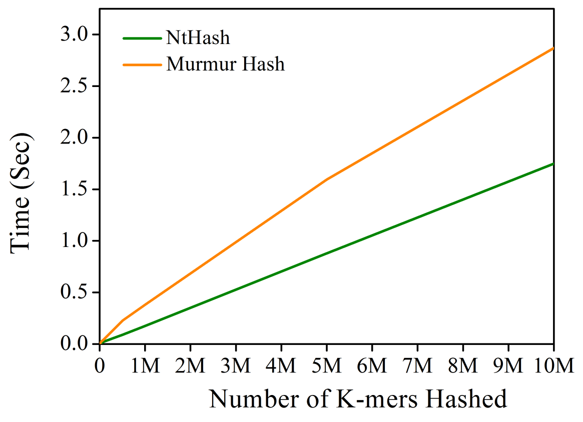

Figure 5. Time Complexity of murmur hash and ntHash

The hash functions were also tested for the insertion, query time, and the false positive probability in the bloom filter. Twelve input files of unique k-mers with k-mer length k=27 were generated and inserted into the bloom filter and another 12 files with the unique k-mers were generated and tested for false positives. The Bloom filter is designed with the theoretical false positive probability of 0.01 for which the number of hashes was set as 6. The Practical false positive probability is recorded for the twelve input files and the FPP is same for both the hash functions which is shown in Table II.

TABLE II. Simulated False Positive Probability of Murmur and Nthash in the bloom filter for unique k-mers

Input Strings	**Murmur Hash**	**Nt Hash**
100000	0.01025	0.01006
500000	0.00995	0.01017
1000000	0.01002	0.01009
2000000	0.01001	0.00505
3000000	0.01007	0.01004
4000000	0.00999	0.01008
5000000	0.01004	0.01005
6000000	0.01008	0.01006
7000000	0.01004	0.01004
8000000	0.01002	0.01008
9000000	0.01006	0.01003
10000000	0.01004	0.01005
10000000	0.01004	0.01005

The Nthash function uses the lookup table for each DNA base which consumes kilo bytes of memory. When the input string is hashed, each hash value is computed using the few arithmetic operations of the seed value extracted from the lookup table. In the case of Murmur hash function, each string is hashed using multiplication and rotation operation which consumes more time for hashing each and every string. When the hash value increases the computation of the murmur hash increases compared to the Nthash function. The time taken for inserting the strings into the bloom filter and querying another set of unique strings from the bloom filter were recorded and shown in Figure 6 and Figure 7. From the above results, we can conclude that for better performance, Murmur hash can be used for the applications using random strings and the Nthash can be used for the genomics applications.

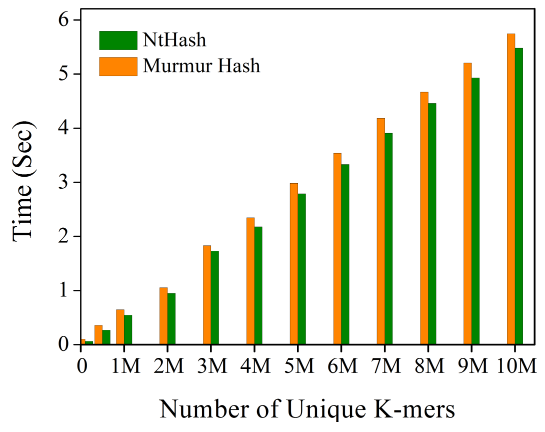

Figure 6. Insertion time of murmur hash and ntHash in Bloom filter

TABLE III. False Positive Probability of Bloom Filter and Partitioned Bloom Filter

FPP (Theory)	No. of Unique Strings	Size of Bloom Filter	FPP (Simulated)	
			Partitioned Bloom Filter	Bloom Filter
0.1	2000000	9585059	0.10274	0.10058
	5000000	23962646	0.10220	0.10064
	8000000	38340234	0.10637	0.10066
	10000000	47925292	0.10233	0.10075
	15000000	71887938	0.10238	0.10070
	20000000	95850584	0.10222	0.10064
0.01	2000000	19170117	0.01617	0.01005
	5000000	47925292	0.01622	0.01004
	8000000	76680468	0.01621	0.01000
	10000000	95850584	0.01611	0.01000
	15000000	143775876	0.01622	0.01004
	20000000	191701168	0.01617	0.01004
0.001	2000000	28755176	0.00807	0.00102
	5000000	71887938	0.00800	0.00102
	8000000	115020701	0.00788	0.00100
	10000000	143775876	0.00799	0.00101
	15000000	215663814	0.00804	0.00099
	20000000	287551752	0.00803	0.00100

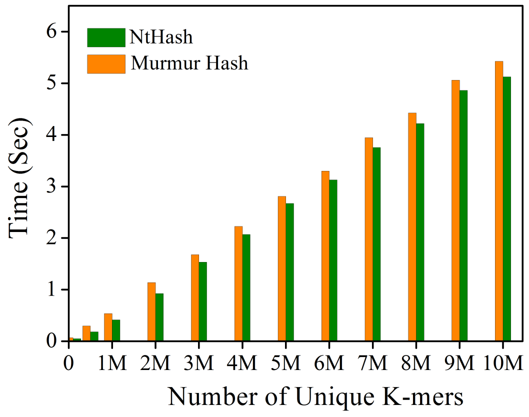

Figure 7. Query time of murmur hash and ntHash in Bloom filter

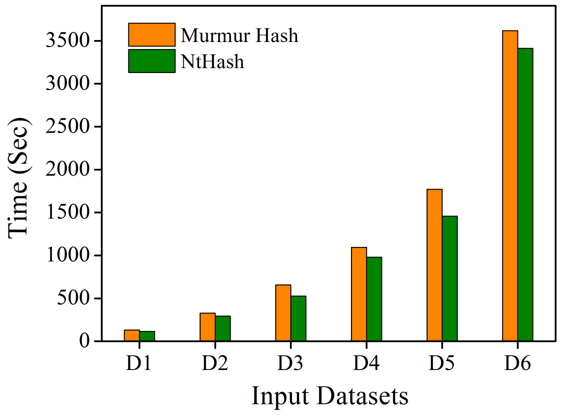

Figure 8. Comparison of Murmur Hash and NtHash in Bloom filter in K-mer Counting process

Since K-mer counting application uses bloom filter, the performance comparison of Bloom filter and the Partitioned Bloom filter is needed. To check the performance of the bloom filter and the partitioned bloom filter, a set of unique random strings were generated and inserted into the bloom filter, another set of unique random strings were generated and tested for the false positive probability in both Bloom and Partitioned bloom filter. The false positive probability of the Bloom and Partitioned bloom filter is shown in Table III. The time taken to insert the strings in the bloom filter and querying the query strings were tabulated in Table IV for the false positive rates of 0.01, 0.1 and 0.001.

From the results, we could see the reduction in the number of memory access of the Partitioned filter. When the input string is hashed, it will give the random location in the standard bloom filter and for h hashes, all the h bits were stored in the random locations in the bloom filter memory. For querying the strings, all the h locations in the entire memory are accessed. In the partitioned bloom filter, all the h bits for an input string were set in a single cache line. Similarly, for the query string only the cache line is accessed. As the number of memory access reduces, time complexity also reduces at the expense of the false positive rate. However, the increase in the false positive rate is very minimal and is under the allowable false positive rate (10 percent error rate is acceptable) of many genome applications genome applications [4], [24].

For testing the K-mer counting process, k-mers with length k equal to 27 were generated from the short reads. The k-mers were hashed and inserted in the partitioned bloom filter and stored in the hash table. The K-mer counting algorithm is implemented in C++ in a multicore environment. The algorithm is parallelized using OpenMP to exploit the parallelism in the available CPU threads. The input files contain the sequence of short reads of the DNA bases, where the average read length of each read varies from 300 to 500. While inserting the k-mer from the read, each k-mer is taken and hashed using Murmur hash and inserted into the bloom filter and then into the hash table. The same process is repeated by using the Nthash function. The time taken to process the input sequence were recorded and shown in Figure 8. Since the Nthash function uses the cyclic hash approach and the lookup table for the DNA bases, the hash computation for the entire dataset is reduced thereby the time taken for the k-mer counting process is reduced.

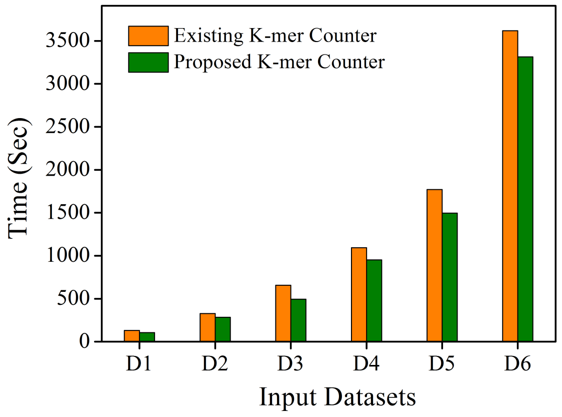

Figure 9. Comparison of Standard Bloom Filter k-mer counter[13] and the proposed hybrid k-mer counter

From the hash and the bloom filter architecture comparison, the Nthash and the Partitioned Bloom filter has better time complexity. Thus, the hash function in the k-mer counting algorithm is replaced with Nthash function and the standard bloom filter architecture is replaced with a Partitioned bloom filter. The set of input datasets were passed to the proposed k-mer counter and the time taken for the entire k-mer counting process was recorded. The same sets of inputs were given to the existing k-mer counter and

TABLE IV. Time Complexity of Bloom Filter and Partitioned Bloom Filter for Random Strings

False Positive Probability	Number of Unique Strings	Size of the Bloom filter	Insertion time		Querying time	
			Bloom Filter	Partitioned Bloom filter	Bloom Filter	Partitioned Bloom filter
0.01	2000000	19170117	1.0506	0.9249	0.6141	0.5847
	5000000	47925292	2.2365	1.9697	1.5358	1.3807
	8000000	76680468	3.4567	3.0577	2.4890	2.2824
	10000000	95850584	4.3045	3.8447	3.1233	2.9229
	15000000	143775876	6.6260	5.7787	4.8583	4.3328
	20000000	191701168	9.6618	7.7818	6.7708	6.0416
0.1	2000000	9585059	0.8810	0.7399	0.6066	0.5460
	5000000	23962646	1.7805	1.6468	1.4881	1.4527
	8000000	38340234	2.7018	2.6512	2.4173	2.1871
	10000000	47925292	3.4245	3.1713	3.0963	2.8011
	15000000	71887938	5.2105	4.8712	4.7199	4.1099
	20000000	95850584	6.5536	6.4608	6.0593	5.5144
0.001	2000000	28755176	1.1374	0.9690	0.6110	0.5581
	5000000	71887938	2.5884	2.1575	1.5479	1.3737
	8000000	115020701	4.0290	3.3455	2.5180	2.2463
	10000000	143775876	5.3573	4.1809	3.4835	2.8774
	15000000	215663814	8.9077	6.4288	5.2295	4.6298
	20000000	287551752	12.6444	8.8052	7.3568	6.4228

the results are shown in Figure 9. From the results, it is evident that there is an improvement in the running time of the proposed k-mer counting process compared to the existing method. The reduction in the computation time is due to the efficient use of the cache memory in the partitioned bloom filter and the reduction in the computation time of the hash function. Since the Nthash is occupying only minimal extra memory for the seed table storage and the partitioned bloom filter architecture is only the modified version of the bloom filter, the increase in the memory utilization is not comparable. As k-mer counting is an initial process in many applications, this reduction in the run time of the process will be beneficial when adopted in applications with huge datasets.

4. Conclusion

In this paper, we proposed a hybrid partitioned bloom filter method for k-mer counting of the DNA sequence of FragariaVesca species. We used the ntHash hashing mechanism for hashing the k-mers present in the short reads of the DNA sequence. The ntHash function used in the hybrid method considerably reduced the computation time compared to the murmur hash function in the existing approach. The partitioned bloom filter aligned to the cache line size reduced the number of memory access to one which in turn reduced the k-mer frequency computation time. The proposed method has been implemented in the multicore environment and simulated with a real-time gene dataset. The computed data for different sets of strings showed that the k-mer counting using a partitioned bloom filter with Nthash took less time compared to the standard bloom filter with Murmur hash. It was also observed that there was a substantial improvement in the processing time when the size of the datasets increased. The proposed hybrid approach will be beneficial to bloom-filter based genome applications. Also, it can be further accelerated using the hardware accelerators such as FPGA and GPU as there is more scope for data decomposition.

References

[1] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: astronomical or genomical?" *PLoS biology*, vol. 13, no. 7, p. e1002195, 2015.

[2] F. Sanger, S. Nicklen, and A. R. Coulson, "Dna sequencing with chain-terminating inhibitors," *Proceedings of the national academy of sciences*, vol. 74, no. 12, pp. 5463–5467, 1977.

[3] F.-D. Pajuste, L. Kaplinski, M. Möls, T. Puurand, M. Lepamets, and M. Remm, "Fastgt: an alignment-free method for calling common snvs directly from raw sequencing reads," *Scientific reports*, vol. 7, no. 1, pp. 1–10, 2017.

[4] S. D. Jackman, B. P. Vandervalk, H. Mohamadi, J. Chu, S. Yeo, S. A. Hammond, G. Jahesh, H. Khan, L. Coombe, R. L. Warren *et al.*, "Abyss 2.0: resource-efficient assembly of large genomes using a bloom filter," *Genome research*, vol. 27, no. 5, pp. 768–777, 2017.

[5] H. Hirakawa, K. Sumitomo, T. Hisamatsu, S. Nagano, K. Shirasawa, Y. Higuchi, M. Kusaba, M. Koshioka, Y. Nakano, M. Yagi *et al.*, "De novo whole-genome assembly in chrysanthemum seticuspe, a model species of chrysanthemums, and its application to genetic and gene discovery analysis," *DNA research*, vol. 26, no. 3, pp. 195–203, 2019.

[6] 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, 2011.

[7] S. Deorowicz, A. Debudaj-Grabysz, and S. Grabowski, "Disk-based k-mer counting on a pc," *BMC bioinformatics*, vol. 14, no. 1, pp. 1–12, 2013.

[8] M. Kokot, M. Dlugosz, and S. Deorowicz, "Kmc 3:counting and manipulating k-mer statistics," *Bioinformatics*, vol. 33, no. 17, pp. 2759–2761, 2017.

[9] M. Erbert, S. Rechner, and M. Müller-Hannemann, "Gerbil: a fast and memory-efficient k-mer counter with gpu-support," *Algorithms for Molecular Biology*, vol. 12, no. 1, pp. 1–12, 2017.

[10] A.-A. Mamun, S. Pal, and S. Rajasekaran, "Kcmbt: ak-mer counter based on multiple burst trees," *Bioinformatics*, vol. 32, no. 18, pp. 2783–2790, 2016.

[11] S. M. Bose, V. S. Lalapura, S. Saravanan, and M. Purnaprajna, "k-core: Hardware accelerator for k-mer generation and counting used in computational genomics," in *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*. IEEE, 2019, pp. 347–352.

[12] S. C. Manekar and S. R. Sathe, "A benchmark study of k-mer counting methods for high-throughput sequencing," *GigaScience*, vol. 7, no. 12, p. giy125, 2018.

[13] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in dna sequences using a bloom filter," *BMC bioinformatics*, vol. 12, no. 1, pp. 1–7, 2011.

[14] S. Saha and S. Rajasekaran, "Ec: an efficient error correction algorithm for short reads," *BMC bioinformatics*, vol. 16, no. 17, pp. 1–10, 2015.

[15] L. Song, L. Florea, and B. Langmead, "Lighter: fast and memory-efficient sequencing error correction without counting," *Genome biology*, vol. 15, no. 11, pp. 1–13, 2014.

[16] Y. Heo, A. Ramachandran, W.-M. Hwu, J. Ma, and D. Chen, "Bless 2: accurate, memory-efficient and fast error correction method," *Bioinformatics*, vol. 32, no. 15, pp. 2369–2371, 2016.

[17] Aappleby, "Smhasher/murmurhash3.cpp at master • aappleby/smhasher," Jan 2016. [Online]. Available: https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp

[18] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[19] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet mathematics*. Citeseer, 2002.

[20] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: A state-of-the-art survey," *Computer Networks*, vol. 57, no. 18, pp. 4047–4064, 2013.

[21] H. Mohamadi, J. Chu, B. P. Vandervalk, and I. Birol, "nthash: recursive nucleotide hashing," *Bioinformatics*, vol. 32, no. 22, pp. 3492–3494, 2016.

[22] Y. Qiao, T. Li, and S. Chen, "Fast bloom filters and their generalization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 93–103, 2013.

[23] F. Putze, P. Sanders, and J. Singler, "Cache-, hash-and space-efficient bloom filters," in *International Workshop on Experimental and Efficient Algorithms*. Springer, 2007, pp. 108–121.

[24] Y. Shibuya, D. Belazzougui, and G. Kucherov, "Space-efficient representation of genomic k-mer count tables," *Algorithms for Molecular Biology*, vol. 17, no. 1, pp. 1–15, 2022.

Elakkiya Prakasam Elakkiya Prakasam is is a research scholar in School of Electronics Engineering at Vellore Institute of technology, Vellore, India. She received her M.E Degree in VLSI Design from Easwari Engineering College, Anna University, India. Her area of research interests includes Multi-core Programming, Parallel Computing and High Performance Computing.

Arun Manoharan Arun Manoharan is an Associate Professor in School of Electronics Engineering at Vellore Institute of Technology, Vellore, India. He has received his Ph.D. degree in High Performance Network Security from Anna University, India. He was a post-doctoral fellow at Institute of Electronics and Telematics Engineering of Aveiro, University of Aveiro, Portugal. His areas of interest include High performance computing and Network security. He has published more than 40 articles in reputed Journals and International Conferences.