Bioinformatics, 2016, 1–8
doi: 10.1093/bioinformatics/btw345
Advance Access Publication Date: 9 June 2016
Original Paper

OXFORD

Sequence analysis

KCMBT: a *k*-mer Counter based on Multiple Burst Trees

Abdullah-Al Mamun, Soumitra Pal and Sanguthevar Rajasekaran*

Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA

*To whom correspondence should be addressed.
Associate Editor: Inanc Birol

Received on January 25, 2016; revised on May 11, 2016; accepted on May 25, 2016

Abstract

Motivation: A massive number of bioinformatics applications require counting of *k*-length substrings in genetically important long strings. A *k*-mer counter generates the frequencies of each *k*-length substring in genome sequences. Genome assembly, repeat detection, multiple sequence alignment, error detection and many other related applications use a *k*-mer counter as a building block. Very fast and efficient algorithms are necessary to count *k*-mers in large data sets to be useful in such applications.

Results: We propose a novel trie-based algorithm for this *k*-mer counting problem. We compare our devised algorithm *k*-mer Counter based on Multiple Burst Trees (KCMBT) with available all well-known algorithms. Our experimental results show that KCMBT is around 30% faster than the previous best-performing algorithm KMC2 for human genome dataset. As another example, our algorithm is around six times faster than Jellyfish2. Overall, KCMBT is 20–30% faster than KMC2 on five benchmark data sets when both the algorithms were run using multiple threads.

Availability and Implementation: KCMBT is freely available on GitHub: (https://github.com/abdullah009/kcmbt_mt).

Contact: rajasek@engr.uconn.edu

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

String algorithms have been frequently used in bioinformatics as genomic sequences can be represented by strings from an alphabet of distinct characters. A substring of length k in a string is defined as a k-mer, where k is a positive integer. k-mers in genomic sequences have been utilized to perform various analyses on the sequences. Numerous applications require counting the occurrences of particular k-mers. A k-mer counter computes the abundance of every unique k-mer in a string or a set of strings. It has become an elementary building block for various bioinformatics applications. Frequencies of k-mers along with the coverage information have been used in assembling genomic sequences (Jaffe *et al.*, 2003; Miller *et al.*, 2008; Pevzner *et al.*, 2001; Zerbino and Birney, 2008), correcting errors in sequencing reads to improve the assembly quality (Kelley *et al.*, 2010; Liu *et al.*, 2013; Medvedev *et al.*, 2011), finding repetitions and solving many other bioinformatics problems (Marçais and Kingsford, 2011).

There exist several efficient k-mer counting algorithms. If the available memory is very large, arrays with a straightforward k-mer indexing can be used for counting. As memory is limited, fast access hashing has become an alternate solution. Most of the currently available solutions engage this approach. Most of the existing algorithms also exploit the fact that the usage of unsigned integers instead of character strings simplifies the problem and facilitates the solution. Spurious k-mers can be easily removed by employing a bloom filter. A bloom filter is a space-efficient probabilistic data structure that can be used to search for the existence of an item in a set. Suffix tree based solutions are also popular.

We introduce a novel internal memory technique called k-mer Counter based on Multiple Burst Trees (KCMBT), which uses cache efficient burst tries to solve this problem. A burst trie is a trie in which a full node is split into multiple nodes to make space for insertion of new elements. This algorithm combines a number of powerful ideas to enable faster output. These ideas include utilization of

burst tries to store k-mers, consideration of $(k+x)$-mers and unifying a k-mer and its count in a single unit. Currently KCMBT is the fastest k-mer counting algorithm. Experimental results in conjunction with theoretical analysis establish our statement.

2 Related works

An obvious approach to k-mer counting is to use a hash table, where any k-mer is used as key and its count as value. If the available memory is very large, a simple array can be used to realize this hashing. On the other hand, simple hashing-based methods with limited memory suffer from large run times. Most of the available algorithms take advantage of multi-threading, out-of-core and locking optimizations to design efficient hash-based solutions.

Tallymer (Kurtz *et al.*, 2008) engages enhanced suffix arrays to store k-mers. But it is computationally expensive, and its memory requirement is highly dependent on the genome size and its coverage.

Jellyfish (Marçais and Kingsford, 2011) introduces a multi-threaded, lock-free hash-table for k-mer counting. It exploits CAS (compare-and-swap) assembly instruction to access and update a memory location in a multi-threaded environment.

Single occurrence k-mers may be produced due to sequencing errors. Bloom filter (Bloom, 1970) is a good way to avoid counting most of these k-mers. Single-threaded BFCounter (Melsted and Pritchard, 2011) uses this probabilistic data structure to store more frequent k-mers in reduced memory. It employs hash tables to store k-mers occurring at least two times. Jellyfish is faster than BFCounter in experimental comparisons.

Sort and compact instead of hashing is another technique to count k-mers. Turtle (Roy *et al.*, 2014) gathers k-mers in an array upto a certain point. Then it sorts and compacts them. This is an in-memory algorithm and the array should be large enough to hold all unique k-mers. Some versions utilize a bloom filter to remove spurious single occurrence k-mers.

Large memory requirements have been further reduced by disk-based methods. DSK (disk streaming of k-mers) (Rizk *et al.*, 2013) counts k-mers with very low memory usage. It partitions k-mers according to hash values computed by some hash functions and stores them in the disk. Later it loads one partition at a time and counts k-mers using a hash table. It computes k-mer frequencies of human genome engaging only 4GB of memory.

Another disk-based parallel k-mer counting algorithm is KMC (K-mer Counter) (Deorowicz *et al.*, 2013). It works in two phases: distribution phase and sort phase. Distribution phases collects k-mers into buffers. When buffers are full, it sends them to different disk files determined by prefixes of k-mers. After storing all the compact k-mers into disks, the sort phase brings back one file at a time, expands k-mers, sorts and merges them to count unique k-mers, and writes them back to the disk. This algorithm is very efficient in terms of time and space.

MSPKmerCounter (Li *et al.*, 2015) is another efficient disk-based k-mer counter. It offers a new technique called Minimum Substring Partitioning to partition reads in a more effective way. Another two phase external memory algorithm is KAnalyze (Audano and Vannberg, 2014). In the first phase, it accumulates k-mers until the allocated memory is full. Then it sorts and merges them, and writes them in the disk. In the second phase, all the disk files are merged in multiple steps.

The most efficient among all of these available k-mer counting algorithms is KMC2 (Deorowicz *et al.*, 2015). It is an extension of the KMC algorithm. It improves by the idea of minimizers to reduce memory and disk space, and the idea of $(k+x)$-mers.

3 Methods

We have devised a trie-based in-memory algorithm, KCMBT, for this k-mer counting problem. For large datasets internal k-mer counting algorithms suffer from huge cache misses. We have found this issue for both hash-based and tree-based solutions. KCMBT shows greater improvement by using cache efficient modified burst tries for storing compact k-mers as well as proper usage of $(k+x)$-mers.

3.1 Burst tries

A burst trie (Heinz *et al.*, 2002; Sinha and Zobel, 2004) is a trie that can be used to store a set of strings efficiently in almost sorted order. The overall trie data structure consists of three components: a set of strings to be placed, an access trie and containers to contain those input strings. Containers can be thought of as leaves of this trie. Each string is matched against branches of the trie until it finds a container or it reaches past the input string. For the later case, there exists a container to store the string or just a terminal symbol or a counter to count such strings. For every internal node the number of children is at most the size of the alphabet. All the strings stored in a container have the same prefix. So the depth of a container is at most the size of the common prefix. Therefore strings without these common prefixes can be placed safely to reduce space requirement.

Insertion of a string into the trie compares the leftmost symbol of the string against branches of the root to find the next level node. Then it chooses the next symbols to move to descendent nodes until it associates with a container. When a container is full, strings of that container are sorted and the container is burst. Then a node with descendent containers is created in place of that parent container and strings of this parent container are distributed among the newly created children containers. The depth of the current container is one plus the depth of its parent container.

Inorder traversal passes containers in sorted order. But strings within a container may not be sorted yet. As the container size is small, a fast radix sort can easily be employed to sort these strings. After a full traversal, we get a set of sorted strings.

We can get a better insight of bursting of a container by following a simple example. Let the container size be 4, and there be a container C having $\{\{CGCC,1\},\{ATGG,1\},\{GTGA,1\},\{CGCC,1\}\}$. Note that the length of any substring in any container will be in general $\leq k$ (since we do not have to store the prefix corresponding to the container). Let the length of the substrings in C be k'. In our algorithm any k-mer with its count is stored in a single 64-bit unsigned integer. Now consider the insertion of another k-mer Q that belongs to the container C whose suffix (of length k') is $CAGG$. As this container is full, we sort and merge k'-mers of the container. As a result, we get $\{\{ATGG,1\},\{CGCC,2\},\{GTGA,1\}\}$, respectively. Then we split this container into three new containers by taking prefix one-symbol of every k'-mer for branching. So the newly generated containers contain $\{\{TGG,1\}\}$, $\{\{GCC,2\}\}$ and $\{\{TGA,1\}\}$, respectively. Consider now the insertion of the k-mer Q. Since this starts with a C, it will be stored in the second container, and the container has now $\{\{GCC,2\},\{AGG,1\}\}$. We see that there is no container for branch T yet. If we get such a k-mer, then we will create a container and store the k-mer in it.

3.2 Compact k-mers

To achieve faster access and computation, k-mers can be considered as binary streams. For this problem, we have only four symbols to keep track of. Therefore two bits are necessary to represent a symbol. We use one 64 bit unsigned integer to hold a full k-mer and its count. Sometimes the number of bits used for the counter may not

be enough for counting the frequencies of some very frequent k-mers. We handle this case in a special way as described in the algorithm section. We use a few thousands of burst trees to ensure that the depths of these trees do not become too large. Having these many burst trees also helps us to compact k-mers more. If $k = 28$, we need 56 bits to store a k-mer. If we use a prefix length of 5 for indexing these tress, then we have 4^5 or 1024 trees. So we only use $(28 - 5) * 2 = 46$ bits for storing any k-mer. We get another 18 bits for storing the k-mer's count. In this manner we are able to store a k-mer and its count together in a 64-bit unsigned integer.

3.3 $(k+x)$-mers

We employ extended k-mers for better performance. An extended k-mer is nothing but a substring of length more than k. If R is any input string of length r, then R has $r - k + 1$ k-mers. If we consider substrings of length $(k + x)$ (for some $k \geq 1$), then R has $r - (k + x) + 1$ $(k + x)$-mers. Each such $(k + x)$-mer is an extended k-mer. We can identify these extended k-mers as follows. While processing any input string find the canonical k-mer or lexicographically minimum of current k-mer and its reverse complement. Then we move to next k-mer by sliding one symbol. We also compute the minimum of this k-mer and its reverse complement. If both of these are from either forward or reverse direction, we express these two canonical k-mers by one $(k + 1)$-mer. If consecutive $(x + 1)$ canonical k-mers are from same direction, a $(k + x)$-mer can be generated by considering this $(k + x)$-length substring as an extended k-mer. Note that two successive k-mers share a $(k - 1)$-mer, and l-successive k-mers share a $(k - l + 1)$-mer. For $k' \geq k$, if a k'-mer has c occurrences, then all of its constituent k-mers will have a count of at least c. If we combine successive k-mers in this way, the total required memory and time will be reduced. As an example, consider a portion of a read sequence *AAGCATA*. If $k = 4$, then the k-mers and their reverse complements in this portion are: {*AAGC*, *GCTT*}, {*AGCA*, *TGCT*}, {*GCAT*, *ATGC*} and {*CATA*, *TATG*}. From each pair we take the canonical one, which is the minimum in lexicographic order. In this example, *AAGC*, *AGCA*, *ATGC* and *CATA* are canonical. *AAGC* and *AGCA* are from the same direction. So we combine them into a $(k + 1)$-mer *AAGCA*. *ATGC* is the reverse complement of a 4-mer of the original portion of the sequence. This will be kept as a single k-mer. The remaining 4-mer *CATA* is also a single k-mer.

3.4 Our algorithm

Our algorithm KCMBT works in three phases. The first phase starts with the insertion of $(k + x)$-mers, the second phase counts $(k + x)$-mers and converts them to k-mers and inserts them into k-mer specific trees, and the last phase employs a final traversal of these k-mer trees to identify all the unique k-mers with their counts. Our algorithm uses burst tries as canonical $(k + x)$-mer containers. Both strands of DNA may be present equally. We do not differentiate between those strands as the input read direction is unknown to us. We consider the lexicographically smaller k-mer between an input k-mer and its reverse complement as a *canonical* k-mer. A burst trie is a trie in which the prefix of a string is matched against branches of the tree corresponding to symbols of a specific alphabet. The suffix of the string is stored in a bucket, which is considered as a leaf of that trie. For this problem we consider only $\{A, C, G, T\}$ as the alphabet. We remove all symbols other than A, C, G and T from the input strings. Then we represent those strings by bit arrays for faster data manipulation. Currently we are using 64 bit unsigned integers to hold 32 symbols, where A is represented by 00, C by 01, G by 10 and T by 11. We do that because currently all available 64-bit machines can operate on 64-bit data. At first the tree has only one bucket, which is populated by $(k + x)$-mers. When the bucket is full, one node is created along with four branches, each branch ends with a bucket as a leaf. MSB 2 bits (prefix one symbol) of $(k + x)$-mers from the parent bucket are matched against edges of that tree to choose the proper bucket. These children buckets only contain suffixes of those $(k + x)$-mers as all the $(k + x)$-mers within a bucket share the same prefixes. In this way, whenever a bucket is full, we split it into four buckets under a node. The size of the buckets are dynamically allocated according to the size of the number of $(k + x)$-mers they get from their parent bucket. In our algorithm, when we split a full bucket of the burst tree, we sort all the $(k + x)$-mers, merge and compact them, and split them into four different buckets.

Our first phase is insertion of $(k + x)$-mers, where x is a predefined value $(0 \leq x \leq 3)$. It reads the input sequences, generates k-mers and their reverse complements, and takes the lexicographically smaller ones. If the current smaller k-mer and the previous smaller k-mer have the same directions, we increment the value of $x' < x$, where x' starts with 0. If the direction changes or $x' = x$, we insert the $(k + x')$-mer into the corresponding tree. To make the process more cache efficient, KCMBT stores $(k + x')$-mers of the same tree into a buffer. When that buffer is full, it inserts all the $(k + x)$-mers of that buffer at a time. The rationale behind using $(k + x')$-mers is that if a $(k + x')$-mer is present c times, then each of its k-mers will be present at least c times. In this way we reduce a huge amount of computation required to insert all those $x' + 1$ k-mers separately.

There are two steps in phase two. The first step traverses every $(k + x')$-mer tree, and counts all the unique $(k + x')$-mers. The second step splits them into k-mers having the same count values, and inserts them into trees of similar prefix k-mer trees. So this phase combines the steps of splitting $(k + x')$-mers and inserting k-mers. After this phase, we have only trees of k-mers.

At the last phase, we traverse all the available trees, and write unique k-mers with their counts to the disk. Some of the k-mers may have counts larger than can be stored in the available bits. We do not merge them at the time of traversal, instead we treat them separately storing them in a structure with their counts.

Algorithm 1 KCMBT

Input: A set of sequencing reads and k (the desired substring length)
Output: A list of unique k-mers in the input with their counts
1: **procedure** KCMBT
2: **for** each read sequence **do**
3: Generate $(k + x)$-mers
4: Insert generated $(k + x)$-mers into corresponding trees
5: **end for**
6: **for** each $(k + x)$-mer tree **do**
7: Traverse the $(k + x)$-mer tree
8: Split the $(k + x)$-mers into k-mers
9: Insert each k-mer into a k-mer tree with the same prefix
10: **end for**
11: **for** each k-mer tree **do**
12: Traverse the k-mer tree
13: Write the k-mers and their counts in the disk
14: **end for**
15: **end procedure**

Table 1. Details of the input data set

Organism	Genome length	No. of bases	Input file size	No. of files	Avg. read length
F.vesca	210	4.5	10.3	11	353
G.gallus	1040	34.7	115.9	15	100
M.balbisiana	472	56.9	197.1	2	101
H.sapiens 1	3093	86.0	223.3	6	100
H.sapiens 2	3093	135.3	312.9	48	101

Genome length in Mbases, no. of bases in Gbases, files size in Gbytes.

The multi-threaded implementation virtually splits each file into t (the number of threads) portions. Each thread reads its assigned part of the sequences, and follows the first two phases of the above algorithm. If every thread constructs n trees, then there will be a total of nt trees. So every prefix has t trees. In the last phase, the trees are shared among the threads. Trees with the same prefixes will be processed by the same thread. Specifically, every thread traverses $\frac{n}{t}$ prefix indexed trees, and accumulates the k-mers in them with their counts.

4 Results

We have compared our KCMBT implementation with previous best-known algorithms. KCMBT has been implemented in C++, and compiled with g++ along with optimization level 3. We ran all the algorithms on a 16 core Dual Intel Xeon Processor E5-2667 machine with 512 GB DDR4 RAM, 12 TB HDD, 256 GB SATA SSD and (Red Hat Linux Enterprise 7.0).

We have collected statistics for Jellyfish-2.2.4, KMC2-2.3.0, Turtle-0.3.1, DSK-2.0.7 and our KCMBT-1.0. We have chosen the latest working implementations of these algorithms. Most of these algorithms perform much better than their original versions published in their respective publications.

As KMC2 is currently the fastest k-mer counting algorithm, we have attempted to compare with it more thoroughly. For this purpose, we have used the same input datasets (Table 1) that KMC2 used for experiments. We have received all the information from their publication. Our input data sets consist of five genomes with varied genome lengths. *Fragaria vesca* is the smallest data set among these five, and *Homo sapiens 2* is the largest one. All of these genomes have multiple compressed fastq files. We have decompressed and concatenated them into one file, so that every tool can handle them easily.

Jellyfish2 requires an initial hash table to store k-mers and their counts. We supplied around 10% more value than the exact unique k-mers count. For example we used 7000M hash size for *H.sapiens 1* where we knew that the total number of unique k-mers is around 6339 millions. Some tools do not count k-mers with single occurrences. We ran all of these implementations in such a way that they output all the k-mers. Turtle is another internal memory algorithm. It has a necessary parameter for the expected number of unique k-mers to select the array size it uses for sorting and counting. We have also used the same number as we have used for Jellyfish2. Turtle comes with three different tools: scTurtle, cTurtle and aTurtle. Each one has two versions to support for maximum 32-mer and 64-mer computations. scTurtle counts k-mers with frequency > 1, cTurtle only reports k-mers with frequency > 1 without showing their counts and aTurtle gives k-mers with all frequencies along with their counts. scTurtle and cTurtle support multiple threads, but aTurle is single threaded. We have used aTurtle for our experiments as it counts k-mers with all frequencies. KMC2 is mainly an external memory or disk-based algorithm, but it has an option to make it an internal memory-based algorithm. Original publication shows results for using 6 GB memory and 12 GB memory limit options. Later option performs better, although we see that it uses less than allocated memory. We have only included results for the 12 GB memory limit option. DSK is another memory frugal disk-based algorithm. It can complete k-mer counting of human genome using only 4 GB of memory. We chose a fixed 6 GB of memory for all the tests although it did not use all of it.

KCMBT is a cache efficient algorithm. It has several parameters which have substantial effects on the running time. Containers of burst trie contain k-mers. If the container size is large, many containers will be partially filled up, and it will leave a huge memory unused. On the other hand, the depth of the trees will be low, and consequently insertion and traversal will take less time. Also there will be less number of sorting, as sorting is called when a container is full. But if the container size is small, memory requirement is low. Sorting and bursting are called frequently, and the height of the tree will be high. Hence insertion and traversal need to compare many branches, and the running time will be increased. Even if the container size is large, consumed time may be high because of numerous cache misses. Another important factor is the number of trees. If we have only one tree, it becomes giant for large genomes. Height of the tree grows fast, and cache misses occur constantly. To keep the depth reasonable, we employ hundreds of trees to store k-mers. Index of the tree for a k-mer is determined by its prefix. As a tree contains k-mers with the same prefix, there is no need to keep that prefix in those k-mers. So we can use this spare bits to manage counting. There are some optimal values for the number of trees dependent on the genome size, cache and memory size. We see in our experiments that 4^5 or 1024 trees work the best for 2 or 4 threads, but 4^6 or 4096 is a good number for a single thread. We chose 4^6 for experiments with a single thread, and 4^5 for multiple threads. We insert k-mers into trees in batches. If we insert one k-mer at a time, caches have to be refreshed possibly each time, which is very time consuming. So we fill a buffer of a fixed size for each tree until it is full. Then we insert these k-mers into that tree. We have found that a buffer size of 1024 is a good value for our experiments. Another major impact factor is the value of x for $(k + x)$-mers. Generation of $(k + x)$-mers is time consuming, but it reduces the total number of insertions or traversals. We use $x = 3$ as a default value as we have received good results with this value.

Fragaria vesca has a comparatively smaller genome size and coverage. We ran Jellyfish2, Turtle, KMC2, DSK and KCMBT. From Table 2, we see that Jellyfish2 is the slowest one among these. Turtle is faster than Jellyfish2, but it has occupied much more memory than Jellyfish2. All of our tests count all the k-mers including single occurrence k-mers. cTurtle and scTurtle do not provide perfect counting. So we used single threaded aTurtle32 for our purposes. DSK is well-known for its careful memory usage. KMC2 is a popular disk-based k-mer counter. It requires almost the same amount of memory as DSK, but it is much faster than DSK. The internal memory version of KMC2 consumes the same amount of time as the external memory one. Our KCMBT implementation shows a remarkable improvement. It is around 50% faster than KMC2 for single thread and two threads, and around 30% faster for four threads. We see that KCMBT is around six times faster than Jellyfish2 for one thread and two threads, and four times for four threads. It uses 14 GB memory for one thread, where 8 GB for two threads and 11 GB for four threads.

We have noticed that KMC2 is more than two times faster when running with four threads than when running with two threads.

Table 2. Results of *k*-mer counters for *F.vesca* ($k = 28$)

Algorithm	One thread			Two threads			Four threads		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	6	0	932	6	0	488	6	0	247
Turtle	14	0	652	—	—	—	—	—	—
KMC2	12	4	298	12	4	188	12	4	84
KMC2RAM	8	0	309	8	0	186	8	0	84
DSK	6	9	392	6	9	211	6	6	112
KCMBT	14	0	160	8	0	89	11	0	59

RAM and disk in GB, time in seconds

Table 3. Results of *k*-mer counters for *G.gallus* ($k = 28$)

Algorithm	One thread			Two threads			Four threads		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	21	0	5536	21	0	2819	21	0	1448
Turtle	56	0	4707	—	—	—	—	—	—
KMC2	12	26	1633	12	26	1037	12	26	445
KMC2RAM	31	0	1657	33	0	1044	33	0	443
DSK	6	48	2871	6	64	1563	6	62	842
KCMBT	42	0	1110	37	0	644	56	0	347

RAM and disk in GB, time in seconds.

Table 4. Results of *k*-mers counters for *M.balbisiana* ($k = 28$)

Algorithm	One thread			Two threads			Four threads		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	11	0	8133	11	0	4013	11	0	2020
KMC2	12	41	2279	12	41	1527	12	41	631
KMC2RAM	47	0	2450	47	0	1510	47	0	646
DSK	6	42	4504	6	48	2502	6	48	1390
KCMBT	37	0	1731	16	0	806	30	0	434

RAM and disk in GB, time in seconds.

KMC2 has two phases. It uses one thread for reading sequences in the first phase. This reading thread does not employ CPU all the time. Therefore it wastes some time here. So when we run with one thread or two threads, the first phase generally uses two threads. One thread is for reading sequences and another one is for computational works. But for four threads there are three threads for computation compared to one in two threads. As a consequence the speedup for four threads is better than for two threads.

Jellyfish2 takes more than 90 min to complete the counting of *k*-mers in *Gallus gallus* for one thread and 25 min for four threads. KCMBT has also performed excellently for this data set, which took 22–30% less time than KMC2. KMC2 needed 445 s for four threads, whereas KCMBT takes 347 s. KMC2(RAM) spends the same amount of time as KMC2 for this genome. DSK is much slower than KMC2. Table 3 compares these tools for *G.gallus*.

aTutrtle32 was run for *Musa balbisiana*, but we waited for several hours without observing any noticeable update, and then we killed the process. So we did not include Turtle for all of our other tests. Deorowicz *et al.* (2015) also showed that Turtle was an underperformer than Jellyfish2 although they included results for scTurtle, which reports *k*-mers with frequency above 1. Table 4 displays results for *M.balbisiana*. Jellyfish2 uses only 11 GB of memory to count *k*-mers for *M.balbisiana*, but it is more than three times slower than KMC2. KCMBT is also around 30% faster than KMC2, and DSK takes more than double the time taken by KCMBT. We see from Tables 2, 3 and 4 that KCMBT for two threads and four threads takes much less memory than for one thread. The memory usage depends on several factors such as *k*-mer distribution of that input genome, percentage of buckets filled up in burst trees, and so on.

Human genome is a massive as well as vital data set. We have compared these tools for *H.sapiens 1* and *H sapiens 2* and arranged the results in Tables 5 and 6, respectively.

Homo sapiens 1 has around 62.7 billion 28-mers and 6.3 billion unique 28-mers. Jellyfish2 is still a very memory efficient internal memory algorithm, as it has used only 41 GB of memory to hold these huge number of unique *k*-mers and their counts in memory. But it is very slow compared to other tools in Table 5. KMC2 is five times faster than this one. KMC2(RAM) achieves a similar speed up. DSK seems to be slow compared to KMC2 and KCMBT. KMC2 is slower than KCMBT by around 35% for one thread, 20% for two threads and 17% for four threads.

Homo sapiens 2 has more coverage than *H.sapiens 1*. The comparative analysis among these tools remains the same from

Table 5. Results of *k*-mer counters for *H.sapiens 1* ($k = 28$)

Algorithm	One thread			Two threads			Four threads		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	41	0	16 433	41	0	8765	41	0	5138
KMC2	12	64	3657	12	64	2351	12	25	1022
KMC2RAM	69	0	3647	70	0	2370	70	0	1030
DSK	6	64	8233	6	70	4490	6	71	2522
KCMBT	70	0	2703	88	0	1952	109	0	875

RAM and disk in GB, time in seconds.

Table 6. Results of *k*-mer counters for *H.sapiens 2* ($k = 28$)

Algorithm	One thread			Two threads			Four threads		
	RAM	Disk	Time	RAM	Disk	Time	RAM	Disk	Time
Jellyfish2	41	0	23 651	41	0	11 938	11	0	6126
KMC2	12	101	5813	12	101	3737	12	101	1816
KMC2RAM	107	0	5584	106	0	3714	107	0	1887
DSK	6	71	15 746	6	70	8753	6	70	4751
KCMBT	82	0	4021	100	0	2433	147	0	1386

RAM and disk in GB, time in seconds.

Table 7. Distribution of consumed time of KCMBT in the three phases (in seconds) ($k = 28$, one thread)

Genome	Insertion	Traversal insertion	Traversal	Total time
F.vesca	95	39	26	160
G.gallus	871	173	66	1110
M.balbisiana	1570	124	37	1731
H.sapiens 1	2205	363	135	2703
H.sapiens 2	3460	420	141	4021

H.sapiens 1 to *H.sapiens 2*. In this case, KMC2 is around 45% slower than KCMBT for one thread and 30% slower for four threads. KCMBT consumes 82 GB memory for a single thread, whereas KMC2(RAM) uses 107 GB. But KCMBT takes 147 GB memory for four threads as every prefix has four trees and similar *k*-mers exist in four trees.

KCMBT works in three phases. The most time consuming phase is the generation of $(k+x)$-mers and insertion of them. We have chosen $0 \leq x \leq 3$ for all of our tests. The first phase generates *k*-mers and their reverse complements, and tracks the minimum or canonical ones. Then it slides one symbol, computes new *k*-mer value and a new reverse complement. It again finds the minimum one of these two. If this minimum and the previous minimum are from the same direction (given a *k*-mer direction or its reverse complement direction), it gets a $(k+1)$-mer. Then it shifts symbol again to extend till $x = 3$ or directions of minimum and previous minimum are different. We maintain fixed length buffers for each tree. Whenever a buffer is full, all the $(k+x)$-mers are inserted into the tree. Insertion is also time consuming. It incorporates traversal to find the proper bucket, and if the bucket is full, a new node and buckets are created and the distribution of $(k+x)$-mers takes place. So we see from Table 7 that 59% of the total time for *F.vesca* was needed for this first phase. This value is 78% for *G.gallus*, 90% for *M.balbisiana*, 81% for *H.sapiens 1* and 86% for *H.sapiens 2*. This phase takes a major portion of the total time, and the ratio increases according to the size of the total number of *k*-mers and unique *k*-mers.

We show the importance of generation and insertion of $(k+x)$-mers in the first phase in Table 8. The last column indicates how many *k*-mers will have to be inserted if we only insert *k*-mers instead of $(k+x)$-mers, where $x > 0$. For each data set, we had to call insertion less than 50% of the times because of the benefits of generation of $(k+x)$-mers. We also see that this phase produces a majority number of $(k+3)$-mers after *k*-mers for every considered genome except *M.balbisiana*. This point is beneficial for the second and the third phases. This reduction in the number of insertions comes at some expense on $(k+x)$-mers generation.

The first phase completes a considerable part of the work. The second phase traverses all the $(k+1)$-mer, $(k+2)$-mer and $(k+3)$-mer trees. After the traversal of a $(k+1)$-mer tree, we get the $(k+1)$-mer count of that specific $(k+1)$-mer. If this count value is *c*, then the constituent two *k*-mers (starting at position 0 and position 1, respectively) will have a count of *c* from this $(k+1)$-mer. $(k+2)$-mer trees contain $(k+2)$-mers, each of which has three *k*-mers. For a $(k+3)$-mer, this value is 4. So when *x* is large, our gain is large as well. Table 9 shows how many unique $(k+x)$-mers exist after the traversal in phase 2, where $1 \leq x \leq 3$. Let c_1, c_2 and c_3 be the counts of $(k+1)$-mers, $(k+2)$-mers and $(k+3)$-mers found in phase 2, respectively. We know that a $(k+x)$-mer covers $(x+1)$ *k*-mers. So we get $2c_1 + 3c_2 + 4c_3$ *k*-mers from only $c_1 + c_2 + c_3$ $(k+x)$-mers, where $1 \leq x \leq 3$. For *H.sapiens 2*, we observe that the average counts of $(k+1)$-mers, $(k+2)$-mers and $(k+3)$-mers is around a billion. This average is more than 65 million for *F.vesca*, 300 million for *G.gallus*, 125 million for *M.balbisiana* and 760 million for *H.sapiens 1*. This improved achievement is the main reason behind the consideration of $(k+x)$-mers. From all of these experiments, we discover that high expenses at first phase are substantially recovered by second and third phases.

Table 10 displays the total numbers of *k*-mers in *k*-mer trees after the first two phases. If the number of *k*-mers in a tree increases, the height of the tree might increase. The other effects are more

Table 8. Generation of $(k+x)$-mers in the first phase of KCMBT ($k=28$, one thread)

Genome	$(k+0)$-mers	$(k+1)$-mers	$(k+2)$-mers	$(k+3)$-mers	Total $(k+x)$-mers	Total k-mers
F.vesca	785	418	258	435	1896	4134
G.gallus	4688	2520	1590	2710	11 508	25 338
M.balbisiana	8067	4322	2607	4133	19 129	41 063
H.sapiens 1	11 743	6303	3945	6639	28 630	62 739
H.sapiens 2	18 565	9807	6226	10 508	45 106	98 893

Number of $(k+x)$-mers in millions.

Table 9. Count of $(k+x)$-mers after traversal in the second phase of KCMBT ($k=28$, one thread)

Genome	$(k+1)$-mers	$(k+2)$-mers	$(k+3)$-mers
F.vesca	72	46	89
G.gallus	364	235	411
M.balbisiana	142	90	145
H.sapiens 1	814	521	955
H.sapiens 2	1007	645	1076

Number of $(k+x)$-mers in millions.

insertion time and traversal time. We have thousands of trees to reduce the height of these trees. The idea of generation, insertion and traversal of $(k+x)$-mers also facilitates our intention of keeping the trees within a reasonable height. The total number of k-mers in k-mer trees is 18 754 millions instead of 62 739 millions for *H.sapiens 1* 26 818 millions instead of 98 893 millions for *H.sapiens 2*. These values are 4–5 times smaller than originally required insertions.

As the inserted numbers of k-mers have been reduced a lot, the time for traversal in the third phase is dramatically reduced. We observe from Table 7 that very little amount of time was spent for this final traversal. Third phase traverses these k-mer trees, and produces k-mers with their counts. If we do not use $(k+x)$-mers for some $x>0$, then we have to traverse larger trees, which is very time taking.

The number of trees has a huge impact on the running time. If we increase the number of trees, the average height of trees decreases. Therefore insertion and traversal take less time. But after a certain number, the running time starts to increase. Because there are already many trees and cache misses occur frequently at the time of insertion. Our observations from experimental results imply that 1024 (4^5) or 4096 (4^6) are quite good numbers for the trees for these data sets. We used 4^6 for all of these experiments. We have included elapsed time for all of these data sets for these two values in Table 11.

We have chosen $x=3$ for all of our experiments. We see from Table 12 that $x=3$ is the optimal value for all of our data sets. For $x=1$ it requires more memory and time than $x=2$ or $x=3$. It employs more memory for $x=5$ than other used values. Large values of x will reduce the number of generated $(k+x)$-mers and the traversal time. But the generation of $(k+x)$-mers will take more time. Also, we have to construct a large number of trees and the cache misses will be frequent after a certain value of x. Our experimental results show that 3 is the optimal value of x.

5 Discussion

KCMBT outperforms every other k-mer counter by a large margin. KMC2 is currently the best-performing k-mer counter. It requires a low internal memory and some not so much inexpensive disk storage. It is practically a very fast k-mer counter with several good options. We have collected data sets of the five genomes used in our experiments from available links in the published paper of KMC2 (Deorowicz *et al.*, 2015). In this KMC2 paper the authors also explain how they ran the other programs. We have tried to employ the same values for the underlying parameters except for the number of cores.

We have shown in the previous section how our ideas were fruitful in counting k-mers efficiently. Our algorithm chooses burst tries to store k-mers. Burst trie is very cache efficient for keeping strings in approximate sorted order. We need that for holding somewhat similar k-mers together. For large genomes, there exist enormous numbers of k-mers. A single tree is not enough to store all of them efficiently. Insertion and traversal become very time expensive operations because of the large height of the tree. We employed hundreds of trees to resolve this issue. When the number of trees is large, we can remove some prefix bits from each k-mer to index its corresponding tree. These extra bits can be used to store counts. Keeping a k-mer and its count together in an unsigned 64 bit word is indeed helpful. Since the alphabet size is 4, each symbol needs at least 2 bits. If we want a 5 symbol prefix, there will be 4^5 or 1024 trees. We ran KCMBT for prefix 0–7 symbols. For 0–4 value, some of these tree heights become large. For a value of 7, there are a massive number of trees, and cache misses occur regularly. We have found good results for values 5 and 6. It is not fruitful to insert one k-mer at a time. If we choose this option, in some of the cases the running time rises more than 25% because of frequent cache swapping. Therefore we choose a large buffer to cache these k-mers and insert them at a time, which is very cache friendly.

Our first attempt was generation and insertion of k-mers. There were two phases. The first phase inserted these k-mers into trees and the second phase traversed them to accumulate all k-mers with their counts. But this idea was not good enough to outperform KMC2 in some cases. We have adapted the very good idea of using of $(k+x)$-mers from KMC2. We have followed almost the same idea to generate these $(k+x)$-mers. KMC2 generates them from super k-mers stored in disk files. We form them from k-mers in the first phase. Experimental outcomes prove the usefulness of this idea. It eliminates many insertion and traverse counts. We generally use $(k+x)$-mers, where $0 \leq x \leq 3$. A value of more than 3 for x works better if they have enough duplicate occurrences. We have noticed that $0 \leq x \leq 3$ performs better in our experiments.

In the second phase, we traverse $(k+x)$-mer trees, and count the number of occurrences of each $(k+x)$-mer. We then split each $(k+x)$-mer into k-mers, and insert them into k-mer trees. We could have improved the time by not inserting the k-mers occurring in the first positions of $(k+x)$-mers. If we did that, we will have to merge at a later stage. The process would be complicated. As the total time spent in the last two phases is not that much, we have avoided this complexity.

Table 10. Insertion of *k*-mers in the first two phases of KCMBT ($k = 28$, one thread)

Genome	Phase 1 $(k+0)$-mers	From $(k+1)$-mers	From $(k+2)$-mers	From $(k+3)$-mers	Insertion total *k*-mers	Total *k*-mers
F.vesca	785	144	138	356	1423	4134
G.gallus	4688	728	705	1644	7765	25 338
M.balbisiana	8067	284	270	580	9201	41 063
H.sapiens 1	11 743	1628	1563	3820	18 754	62 739
H.sapiens 2	18 565	2014	1935	4304	26 818	98 893

Number of $(k+x)$-mers in millions.

Table 11. Distribution of consumed time of KCMBT for different number of burst trees ($k = 28$, one thread)

Genome	1024	4096
F.vesca	153	160
G.gallus	1328	1110
M.balbisiana	1585	1731
H.sapiens 1	3007	2703
H.sapiens 2	5409	4021

Time in seconds.

Table 12. Performances for different values of *x* used for $(k+x)$-mers in KCMBT ($k = 28$, four threads)

Genome	$x=1$		$x=2$		$x=3$		$x=4$	
	RAM	Time	RAM	Time	RAM	Time	RAM	Time
F.vesca	12	69	11	61	11	59	12	61
G.gallus	60	424	55	389	56	347	62	384
M.balbisiana	34	550	30	479	30	434	38	480
H.sapiens 1	114	1017	109	919	109	875	126	973
H.sapiens 2	175	1670	149	1467	147	1386	180	1667

RAM in GB, time in seconds.

6 Conclusions

Efficient *k*-mer counting algorithms are crucial in solving many bioinformatics problems effectively. A considerable number of in-core and out-of-core algorithms are available for this problem. In this article we have offered several original ideas to generate faster output. Usage of multiple burst trees, incorporation of $(k+x)$-mers, splitting and merging these $(k+x)$-mers into *k*-mers, handling a *k*-mer along with its count and several other ideas enable our KCMBT algorithm perform better than the other available algorithms. KCMBT outperforms all the best-known algorithms including Jellyfish, Turtle, and KMC2. Our implementation takes a considerable amount of memory. We are working on reducing the requirement of huge memory for any number of threads. We are also generating ideas to make this algorithm out-of-core so that personal computers with a limited memory can run it for any genome.

Funding

This research has been supported in part by the following grants: NIH R01-LM010101 and NSF 1447711.

Conflict of Interest: none declared.

References

Audano,P. and Vannberg,F. (2014) KAnalyze: a fast versatile pipelined k-mer toolkit. *Bioinformatics*, **30**, 2070–2072.

Bloom,B.H. (1970) Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, **13**, 422–426.

Deorowicz,S. *et al.* (2013) Disk-based k-mer counting on a PC. *BMC Bioinformatics*, **14**, 160.

Deorowicz,S. *et al.* (2015) KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, **31**, 1569–1576.

Heinz,S. *et al.* (2002) Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst. (TOIS)*, **20**, 192–223.

Jaffe,D.B. *et al.* (2003) Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome Res.*, **13**, 91–96.

Kelley,D.R. *et al.* (2010) Quake: quality-aware detection and correction of sequencing errors. *Genome Biol.*, **11**, R116.

Kurtz,S. *et al.* (2008) A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, **9**, 517.

Li,Y. *et al.* (2015). MSPKmerCounter: a fast and memory efficient approach for k-mer counting. *ArXiv e-prints*, 1505.06550.

Liu,Y. *et al.* (2013) Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics*, **29**, 308–315.

Marçais,G. and Kingsford,C. (2011) A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, **27**, 764–770.

Medvedev,P. *et al.* (2011) Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, **27**, i137–i141.

Melsted,P. and Pritchard,J.K. (2011) Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, **12**, 333.

Miller,J.R. *et al.* (2008) Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, **24**, 2818–2824.

Pevzner,P.A. *et al.* (2001) An Eulerian path approach to DNA fragment assembly. *Proc. Natl Acad. Sci. USA*, **98**, 9748–9753.

Rizk,G. *et al.* (2013) DSK k-mer counting with very low memory usage. *Bioinformatics*, **29**, 652–653.

Roy,R.S. *et al.* (2014) Turtle: identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, **30**, 1950–1957.

Sinha,R. and Zobel,J. (2004) Cache-conscious sorting of large sets of strings with dynamic tries. *J. Exp. Algorithmics (JEA)*, **9**, 1–5.

Zerbino,D.R. and Birney,E. (2008) Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, **18**, 821–829.