Interdisciplinary Sciences: Computational Life Sciences (2020) 12:99–108
https://doi.org/10.1007/s12539-019-00348-5

ORIGINAL RESEARCH ARTICLE

Counting Kmers for Biological Sequences at Large Scale

Jianqiu Ge[1] · Jintao Meng[1] · Ning Guo[1] · Yanjie Wei[1] · Pavan Balaji[2] · Shengzhong Feng[1]

Received: 8 July 2019 / Revised: 19 August 2019 / Accepted: 25 October 2019 / Published online: 16 November 2019

Abstract
Counting the abundance of all the distinct kmers in biological sequence data is a fundamental step in bioinformatics. These applications include de novo genome assembly, error correction, etc. With the development of sequencing technology, the sequence data in a single project can reach Petabyte-scale or Terabyte-scale nucleotides. Counting demand for the abundance of these sequencing data is beyond the memory and computing capacity of single computing node, and how to process it efficiently is a challenge on a high-performance computing cluster. As such, we propose SWAPCounter, a highly scalable distributed approach for kmer counting. This approach is embedded with an MPI streaming I/O module for loading huge data set at high speed, and a counting bloom filter module for both memory and communication efficiency. By overlapping all the counting steps, SWAPCounter achieves high scalability with high parallel efficiency. The experimental results indicate that SWAPCounter has competitive performance with two other tools on shared memory environment, KMC2, and MSPKmerCounter. Moreover, SWAPCounter also shows the highest scalability under strong scaling experiments. In our experiment on Cetus supercomputer, SWAPCounter scales to 32,768 cores with 79% parallel efficiency (using 2048 cores as baseline) when processing 4 TB sequence data of 1000 Genomes. The source code of *SWAPCounter* is publicly available at https://github.com/mengjintao/SWAPCounter.

Keywords Kmer counting · Biological sequence · Counting bloom filter · Scalability

1 Introduction

Advancing breakthroughs in Next-Generation Sequencing (NGS) have caused an unprecedented increase of high-quality sequencing data. Representative Sequencing platforms such as Illumina X Ten, SOLiD Ion GeneStudio S5 series, and Roche MagNA Pure 96 System now can produce longer high-throughout sequencing reads with lower price. However, the growth rate of biological sequencing data has become faster and faster, and it poses a great challenge to the design and development of modern bioinformatics tools [1, 2].

A critical step of the sequence pre-analysis in many bioinformatics pipelines is kmer (sub-string of length k) counting, and it has become an essential component to those applications. For de novo genome assembly, most popular parallel genome assemblers are based on de Bruijn graph [3, 4]. A throughout overview of the number and the distribution of distinct kmers provide a detailed statistical information about the characteristics of this kind of graph. Moreover, the counting results provide a reference to discard low-abundance kmers, which are highly likely caused by sequencing errors. Thus, building the histogram of frequency for each distinct kmer is dramatic for various kinds of applications, including for de novo assembly [5–7], error correction [8–11], multiple sequence alignment [12], and metagenomic data classification and clustering [13].

Despite the progress in NGS platforms, many types of sequencing errors exist, including substitutions, insertions, and deletions [10]. These will generate large amount of erroneous vertices and edges when assembling short-sequence reads into the whole genome. Moreover, sequencing errors also make the kmer counting problem more complex. Most common sequencing machines such as Illumina X Ten

Jianqiu Ge and Jintao Meng contributed equally.

✉ Yanjie Wei
yj.wei@siat.ac.cn

[1] Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Beijing 518055, China

[2] Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439-4844, USA

introduce approximately 0.1–1% sequencing error. From Fig. 1, one can see the statistics of Yanhuang data set (300 GB, 100× coverage) as a function of the length of kmer. It depicts that even such a low sequencing error rate could result in about 60% erroneous kmers. As a result, the intermediate data size of storing kmers in memory can be dramatically larger than the input data size.

Recently there has been much effort to develop tools for sequence analysis based on kmer counting. State-of-the-art kmer counting tools can be classified into two categories: one is shared memory tools with multiple threads and the other is distributed tools based on distributed memory environment. The widely used shared memory kmer counting tools include Jellyfish [14], MSPKmerCounter [15, 16], BFCounter [17], KMC2 [18], DSK [19], KHMer [20], and Turtle [21]. Furthermore, we can classify these tools into three categories [22]. The first are hash table with lock-free-based tools, such as Jellyfish and BFCounter. The second are bloomfilter-based tools, such as BFCounter, Turtle, and KHMer.

Last are partitioning-based tools, the representative tools in this category are KMC2, MSPKmerCounter, and DSK.

Several parallel tools take advantage of high-performance clusters to lift the limit of memory usage and successfully accelerate the whole process. The representative ones include Kmerind [23], bloomfish [24], and HipMer [25, 26]. Kmerind is a flexible and extensible distributed kmer counting tool and indexing library, which is implemented on MPI. Bloomfish is another scalable kmer counting method which makes use of Jellyfish building on top of a MapReduce framework-Mimir [27]. HipMer is a highly parallelized de novo assembler that includes a similar kmer counting module. More details will also be discussed in the Sect. 5.

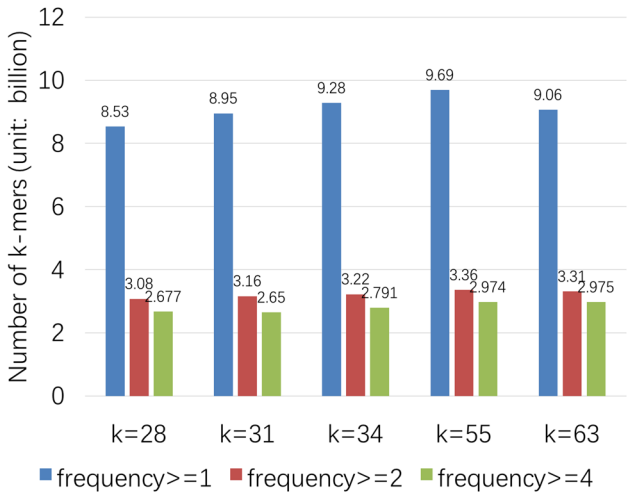

Fig. 1 The effect of sequencing error on distinct kmer's frequency distribution of Yanhuang data set. The kmers with its frequency less than and equal to 3 can be recognized as kmers generated by sequencing error

Traditional kmer counting tools based on single node are not highly parallelized, which are not able to handle biological sequencing data as it scales up to Terabytes or even Petabytes. Existing kmer counting tools with distributed memory have limited scalability. On a single computing node with limit resource environment, these distributed tools rarely outperformed shared memory tools especially KMC2. The same efficiency issues also occur in I/O part and communication part, which affect the overall performance of these methods.

In this paper, we propose a novel distributed kmer counting tool named SWAPCounter. It also delivers a competitive performance with current state-of-the-art single-node kmer courting tools such as KMC2 and MSPKmerCounter. To our knowledge, this work is the first kmer counting tool that scales to 32,768 cores. The contribution of our work are threefold:

1. An MPI streaming I/O module is designed with caching mechanisms and data pooling technique to utilize I/O peak performance.
2. Non-blocking all-to-all communication is used to overlap the computation and communication to improve the performance and efficiency.
3. In the optional step of kmer filtering, we use counting bloom filer to discard low-abundance kmers. As a result, it saves memory consumption with negligible counting errors and false positives.

The rest of the paper is organized as follows. Section 2 gives a brief introduction on kmers and counting bloom filter. Section 3 describes the design and algorithm of SWAPCounter. Section 4 analyzed and evaluated the scalability and performance of SWAPCounter. Section 5 presents and analyzes previous works on kmer counting. The conclusion is summarized in Sect. 6.

2 Preliminaries

2.1 Subsequence-kmer

DNA sequence is a string consisting of A, C, G, T alphabet. A kmer is a sub-string with length k extracted from a given DNA sequence. Figure 2 shows a sliding window of width k cutting a given DNA sequences (length L) into continuous sub-sequences of length k, and finally, the given DNA sequence can be parsed into $L - k + 1$ kmers.

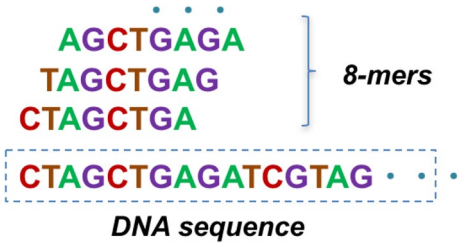

Fig. 2 An example illustrating the kmer extraction process with a given one DNA sequence, and here, the length of kmer k is set to be 8

2.2 Counting bloom Filter

Bloom filter [28] is a space-efficient probabilistic data structure, which can be used to check whether an element is in a set or not. It provides a representation of a set and allows false-positive query results with an extremely low probability and ensure no false negatives. Elements can be added to the set, but cannot be removed (though this can be addressed with a "counting" bloom filter); the more elements that are added to the set, the larger the probability of false positives.

The basic data structure of a bloom filter consists of a bit vector of length m and t hash functions. Each empty cell in that table represents a bit, and the number below is its index. To add an element to the bloom filter, we simply hash it t times and set the bits in the bit vector at the index of those hashes to 1 [29]. Figure 3 shows an example of classical bloom filter with vector length of 18 and 3 hash functions for a given set of variables $\{x, y, z\}$, the query results of w shows that w is definitely not in this set. With a bloom filter of m bits, t hash functions, and n inserted variables, the exact probability of false positives is listed in Equation $\text{Prob} = (1 - (1 - \frac{1}{m})^{tn})^t = (1 - e^{-tn/m})^t$ [28]. According to this equation, the probability of false positives decreases as m (the number of bits in the array) increases, and increases as n (the number of inserted elements) increases.

A counting bloom filter (CBF) is a variant of bloom filter. Compared with bloom filter, a counting bloom filter keeps an array of m counters instead of bits; the counters can track the frequency value for each element currently hashed to its corresponding index. To avoid overflow, each counter will keep its max value from increasing. For our kmer counting application, the maximum value of each element storing with z bits in CBF is $2^z - 1$. Here, $2^z - 1$ should be larger than θ, where θ is the kmer filtering threshold given by user. Figure 4 shows the concept of counting bloom filter implemented in our tool.

3 System Design and Implementation

SWAPCounter consists of four components, parallel sequence I/O, kmer extraction and distribution, kmer filtering with CBF, and counting and statistics. These four components are illustrated in Fig. 5 and will be described in the following subsections. The most time-consuming parts are the first three components. To speed up the implementation of the whole workflow, these three phases are further overlapped with data streaming technology of the entire pipeline.

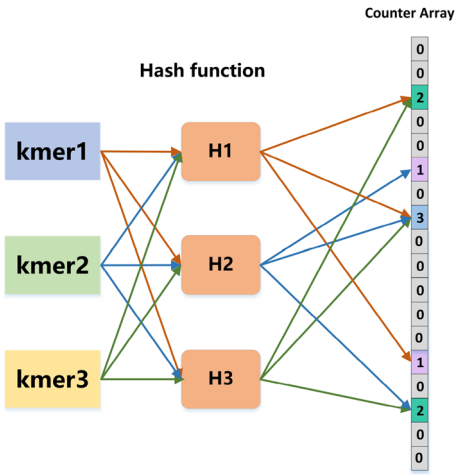

Fig. 4 An example illustrates the data structure and work flow of counting bloom filter. In this example, the counting bloom filter is consisted with a vector of length 18 and 3 hash functions. If the elements in counters are 1-bit size, the counting bloom filter will be degraded into a classical bloom filter

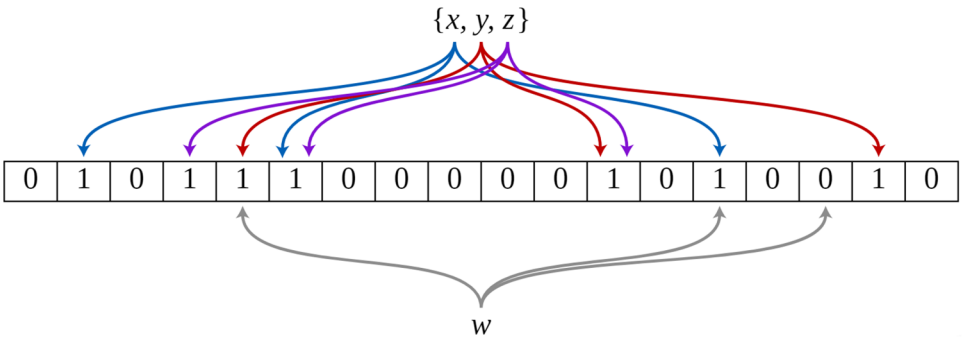

Fig. 3 An example illustrates the data structure and work flow of classical bloom filter

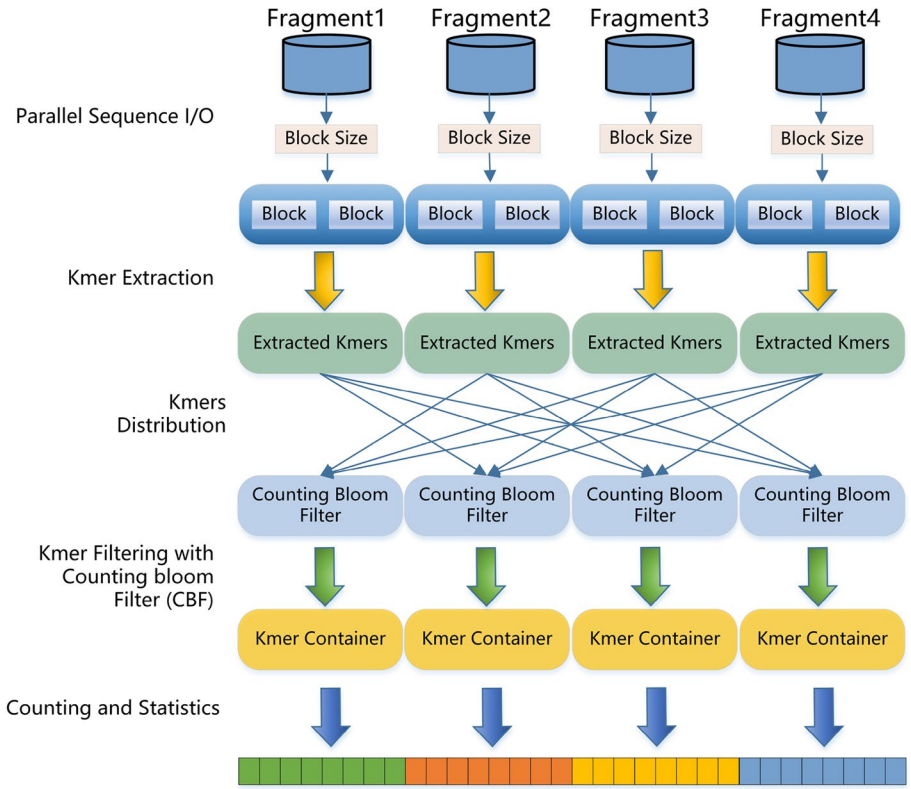

Fig. 5 Workflow of kmer counting in SWAPCounter

3.1 Parallel Sequence I/O

For many supercomputing applications, there is a big gap between the theoretical and practical performance on I/O part. For such purposes, we propose a highly scalable I/O module, which is built on top of MPI I/O, taking advantage of collective I/O optimization, resulting in a strongly I/O performance boost.

Two factors affect the performance of I/O module. The first is data splitting. Traditionally, we split the input data evenly into p (number of processes) virtual data blocks, but the sequence reads are format-sensitive, and splitting points are possibly in the middle of sequence reads. The second is reading data bit by bit which cause less efficient loading, and thus, the computational complexity is bounded by $O(n/p)$, where n and p are the number of characters and processes. Therefore, to couple the system mechanisms to boost performance with MPI I/O is a big challenge.

To overcome the first drawback, we propose an efficient locating function. Each processor with rank ProcessID can quickly slide to the start point of next read. The previous partial read will be joined to its beginning partial read by sending the break position message to assist neighboring process ProcessID − 1 to update ending position. Second, caching is used for I/O optimization. The core concept of cache mechanism is to use MPI I/O functions with file system data blocks, and the size of file system data block equals the default size of a cache page. After taking this strategy, the computational complexity can be bounded by $O(n/(p \times \text{block size}))$. Specifically, by keeping block size as a adjustable parameter, the peak I/O performance can be maximized to approach the system peak.

Furthermore, due to the overlapping between parallel sequence I/O step and communication in kmer extraction and distribution step, SWAPCounter takes advantage of data pooling technique to isolate these two phases and mitigate the latency problem between these two steps. As Fig. 5 shows, we provide a shared memory space (block buffer in Fig. 5) as a data pool, so that the I/O phase can constantly send sequence data to this block buffer, and the step of kmer extraction and distribution can continuously read sequences without interruptions. Thus, the data pool can be regarded as a queue, and the I/O and the kmer extraction/distribution step represent as producer and consumer, respectively. When the data pool is in an appropriate size, these two steps can maintain a balance.

3.2 Kmer Extraction and Distribution

After receiving the sequences from parallel sequence I/O step, each process needs to extract kmers from sequences and then distributes it to their corresponding processes according to a given hash function.

In kmer extraction phase, a sliding window of length k will be applied to cut each sequence into continuous subsequences of length k (or what we called kmers). The computation complexity of this step is bounded by $O(\frac{n\times(L-k+1)}{p})$, where L is the average length of the sequence reads.

In kmer distribution phase, a two-level hash is used to distribute and store kmers. The first-level hash is used to map a kmer to certain processor, and the second-level hash is responsible for mapping kmers to the physical address in memory where we should store it. All generated kmers are collected and grouped with the first-level hash function. These grouped kmers will be sent to their "own" process with all-to-all collective communication primitives, and finally stored in a second-level hash table. The expected computation complexity and communication complexity of this step are also $O(\frac{n\times(L-k+1)}{p})$.

In the above two phases, two techniques can be applied to further improve the computation and communication efficiency. The first one is data pre-compression and instruction-level optimization on kmer extractions. In this method, the bit operation and SSE instruction are used to further compress the computation time in this phase. The second technique is to overlap the computation time in kmer extraction phase and the communication time in distribution phases. In this method, the non-blocking all-to-all communication optimization helps to improve the efficiency of the whole step.

3.3 Kmer Filtering with Counting Bloom Filter

In this section, we aim at implementing an approach to filter erroneous kmers with counting bloom filters. This methodology accelerates kmer counting and also provides a memory efficient filtering process for both kmer storage and modification. Furthermore, since almost all the low-abundance kmers are discarded with CBF, the remaining counting size of kmers is reduced as much as 60%. With far less kmers, the total communication volume and computation on statistics are also reduced.

The essential concept of counting bloom filter used in SWAPCounter is illustrated in Algorithm 1. All the counting bloom filters are initialized with each counter set to 0. t hash functions are designed, and the counters track the frequency value of kmers currently hashed to corresponding location. Then, for each kmer, we check whether it has been inserted in the hash table or not. A kmer container is reserved to store trustworthy kmers. A kmer in CBF can and only can be treated as a trustworthy kmer and stored into the reserved container until all the counters' values calculated by t hash functions are equal to a given filtering threshold θ. If not, all these t counters will be increased by one or keep the max value of the given threshold θ. Compared with existing tools, SWAPCounter is more flexible for kmer filtering and can be used to handle various filtering threshold instead of 1.

After filtering, all trustworthy kmers are grouped into kmer container. A distributed hash table is constructed as that container to store kmers with its frequency larger than a given filtering threshold θ. For all the kmers stored in hash table, it is recorded as < key, value >, where the keys are the kmers and the values are the corresponding occurrence of kmers. Besides, each kmer is stored in a particular processor, so each processor is responsible for a local hash table to store targeting kmers. Furthermore, all these local hash tables are independent.

The use of counting bloom filter also has a trade-off between memory and the false-positive rate, while counting bloom filter and the hash table relative operation is another trade–off. When the cut-off threshold is set bigger than one, it represents we do a kmer binning operation for each trustworthy kmer before inserting them into hash table. Thus, compared with common kmer counting without counting bloom filters, not only the size of hash table and communication reduced, but also the efficiency of insertion improved. Besides, since the counting bloom filters consume significant memory, instead of allocating a counting bloom filter to each core, we choose to allocate one to each compute node. This optimization achieves at least 4× memory consumption.

Algorithm 1: kmer filtering algorithm with counting bloom filter.

```
Input: Given an array A of kmers, and a filtering threshold θ.
Output: Counting the frequency of kmers in the input array
Initialize a CBF with ⌈ log(θ) ⌉ bits for each counter ;
Initialize an array B for hash functions ;
for kmers ∈A do
    if isNotCanonical(kmer) then
        RevComp(kmer) ;
    end
    Min = 0 ;
    for Hash ∈B do
        Pos_i = Hash(kmer) ;
        CBF(Pos_i) = min(CBF(Pos_i)+1, θ) ;
        Min = min(Min, CBF(Pos_i)) ;
    end
    if Min ≥ θ then
        Insert kmer into trustworthy container ;
        And counting its frequency from θ ;
    end
end
```

3.4 Counting and Statistics

Reverse Complement: since the extracting direction of DNA sequences is rarely known, forwards and backwards share the same possibility. In the process of genome assembly, assemblers treat the original kmers and their reverse complements (A< − >T and C< − >G) as equivalent, they represent the same distinct node in de Bruijn graph. Thus, we also follow this rule in SWAPCounter.

The goal of kmer counting is to build the histogram of abundance of all the distinct kmers. Two types of counting results consist of the essential components. One is the number distinct kmers; the other is frequency of each distinct kmer. Relying on these two types of results, almost all the other counting results can be computed.

4 Experiments and Evaluation

4.1 Evaluation Platform and Data Sets

In this section, we evaluate the performance of SWAPCounter both on a *single node* and *cluster*. Single-node tests are conducted on a Knights-Landing (KNL) server. It contains a XeonPhi 7210 with 64 cores of 1.3 GHz and 128 GB DDR3 RAM. Cluster experiments are performed on an IBM Blue-Gene-Q supercomputer named as Cetus at Argonne National Laboratory. Cetus contains 4 cabinets, 4096 compute nodes. Each compute node is equipped with 16 IBM PowerPC A2 processors and 16 GB DDR3 RAM. All the compute nodes are connected by Infiniband. The network of Cetus follows 5D-torus Architecture. The I/O part of Cetus uses IBM GPFS system which supports parallel I/O, and the theoretical peak performance of its I/O bandwidth is 32 GB per second per rack. In terms of data sets, Hg14, Yanhuang and 1000 Genomes in Table 1 are chosen to be evaluated in our experiment.

4.2 Single-Node Comparison

First, we conduct a performance evaluation on a single Knights-Landing (KNL) server with 64 cores. MSPKmerCouter and KMC2 are chosen as another two kmer counting tools for the comparison. The smallest data set in Table 1, Hg14 data set, is used in this experiment. We assigned four threads per core when running these three tools including SWAPCounter, the number of threads increases from 16 to 256. The experimental results are collected and presented in Table 2.

According to Table 2, one can see that the time usage decreases with the increasing number of threads from 16 to 64. When the number of threads continuously increases from 64 to 256, SWAPCounter achieves a sustained acceleration with 4.5X speedup and can scale up to 256 threads, while KMC2 and MSPKmerCounter have a slightly performance decline and achieve negligible speedup of 1.06× and 1.12×, respectively. Communication and thread synchronization overhead are accounting for KMC2 and MSPKmerCounter's poor performance. KMC2 shows the best overall performance at 64 cores. However, it is not a scalable method for kmer counting, as its performance degrades when the number of cores was beyond 64.

Table 1 Detailed statistics of Experiment data

Data set	Hg14	Yanhuang	1 k Genome
Ref size (GB)	0.0886	3	3
Data size (GB)	14.2	300	4096
Read length (bp)	101	80–120	100

Table 2 Time usage (s) of kmer counting test on a single KNL server with Hg14 data set

Treads	16	32	64	128	256
SWAPCounter	182.2	91.3	63.6	49.1	40.3
KMC2	43.5	38.2	36.2	40.3	40.8
MSPKmerCounter	62.0	52.2	48.2	48.7	55.1

4.3 The Effect of CBF

Two advantages benefit kmer counting when applying counting bloom filter (CBF). First, CBF can be used to filter kmers below a given threshold θ before actually storing them. Second, no false positives will happen in CBF, and the effect of false positives are also limited and acceptable in the kmer counting application. An experiment is conducted to evaluate the effect of CBF on different types of data set. Hg14 data set is prepossessed with error correction tool Quake [8], Yanhuang data set is a 100× coverage sequencing data set, and the final 4 TB 1000 Genome data set is randomly selected from 1000 human genome sequencing project. According to Table 3, we can see that percent of kmers with false counted frequency caused by CBF is about 0.001%, 0.01%, and 0.56% respectively. For all these kmers with false counted frequency, these average distances of the rightly and the wrongly counted frequency is about 1.618, 1.160, and 1.242, respectively. The above two results show that there is only a tiny difference on the percentage of wrongly counted kmers and their counted frequency, and will not cause negative results on post-processes, such as Genome Assembly, as no kmers are missing for using CBF.

With the above three data sets, we can find that the kmers with its frequency ≥ 5 occupy a percentage of 98% (erroneous kmers are corrected by error correction tool), 29%, and 3.8% respectively. When we turn to the memory usage affects of CBF, the memory usage with and without CBF for Hg14 is 5 GB and 4 GB, for Yanhuang data set is 656 GB and 148 GB, and for 1 k genome is 4.7 TB and 1 TB. Thus, for the data set without error correction tool, CBF can save more than four times memory usage compared with these kmer counter with no CBF support. That means, by applying CBF in kmer counting, one can save approximately four

Table 3 The statistics and memory usage effects on applying counting bloom filter (CBF) in kmer counting with Hg14, Yanhuang, and 1 k genome data set

	Hg14	Yanhuang	1000 genome
No. kmers with			
False frequency	9681.00	993,151.00	530,365,164.00
Avg freq distance	1.62	1.16	1.24
No. kmers with			
Freq ≥ 5 (Billion)	0.83	2.61	3.63
No. all kmers			
(Billion)	0.85	8.97	94.47
Memory usage			
Without CBF (GB)	5.00	656.00	4700.00
Memory usage			
With CBF (GB)	4.00	148.00	1094.00

In the following table, the number of processes used by Hg14, Yanhuang, and 1 k Genome data set are 16, 1024, and 4096, respectively. Thus, the memory usage is counted as the sum of all processes' resident memory usage (RES). Note that the parameter of reference size given for SWAPCounter for these three data sets is 88.6 MB, 2.6 GB, and 3.0 GB, the kmer size, and cut-off threshold given to all data are 31 and 5

Table 4 Strong scaling evaluation results for SWAPCounter on Cetus (time: s)

Cores	Hg14	Yanhuang	1 k genome
256	14	1323	–
512	12	676	–
1024	6	353	–
2048	4	189	2651
4096	12	96	1352
8192	13	61	826
16384	49	33	411
32768	–	19	208

times memory usage at the risk of a few acceptable false positives.

4.4 Strong Scaling Evaluation

To evaluate the scalability of SWAPCounter, we conducted a strong scaling test on Cetus. The number of CPU cores increases from 256 to 32768, which is the largest number of cores we can use in Cetus. The k value is set to 31. The results are shown in Table 4 and Fig. 6.

In Table 4, one can see that SWAPCounter can scale to 2048 cores and analyze the Hg14 data set in 4 s with 2048 cores. This is nine times faster than KMC2 on the Knights-Landing server. Figure 6 depicts that when processing 300 GB Yanhuang data set, SWAPCounter can scale to 32,768 cores and achieve a speedup of 69.63×, with 54% parallel efficiency (256 cores as baseline). For 4 TB randomly selected genomic data from 1000 Genome project, it is so large that the time usage has reached the maximum time slot (1 h) allowed by the system when the core number is less than 1024. Thus, for 1000 Genome data set, the experiment starts from 2048 cores. Figure 2 shows that SWAPCounter can also scale to 32,768 cores, and reach 12.7× speedup with 79% parallel efficiency (2048 cores as baseline).

Scalability: Figure 6 confirms that SWAPCounter can scale from 256 cores to 32,768 cores on both 300 GB Yanhuang data set and 4 TB data selected from 1000 Genome project.

Speedup: The above strong scaling tests show that SWAPCounter achieves 69.63× speedup when the number of cores scales from 256 to 32,768 on Yanhuang data set, and 12.7× speedup when the number of cores scales from 2048 to 32,768 on 4 TB data from 1000 Genome project.

Efficiency: With the above two data sets, SWAPCounter achieves an efficiency of 54% and 79%, respectively, on Cetus with up to 32,768 cores.

Regrading to other distributed kmer counting tools, an indirect comparison and discussion is conducted. Since both Kmerind and Bloomfish cannot install successfully in Cetus supercomputer, we take results from published works for comparison. Both bloomfish and kmerind show a lower scalability (3072 as the largest) than SWAPCounter. In strong scaling test of bloomfish, when processing HG00096 data set from 1000 Genome on Comet at San Diego Supercomputer Center, it achieves 6.1× speedup with 19% parallel efficiency as the number of cores increases from 24 to 768. For kmerind, it shows a 11× speedup with 69% parallel efficiency as cores double from 16 to 1024 when processing 7.5 GB Cornfield Soil Metagenomic data set.

5 Related Works

In this section, we mainly refer to eight representative kmer counting tools (five for single-node tools and three for distribute environment) to present a brief discussion on each of them.

1. *Jellyfish* makes use of a lock-free hash table and a lock-free queue for communication, so that several threads can update the hash table at the same time. Compared with the traditional methods, it saves much time using more efficient compare-and-swap (CAS) operation. In addition, one should be noted that Jellyfish must set the parameter to estimate the number of distinct kmers. If the predicted value and actual value differ greatly, it will cost enormous extra I/O operation and induce much longer execution time.

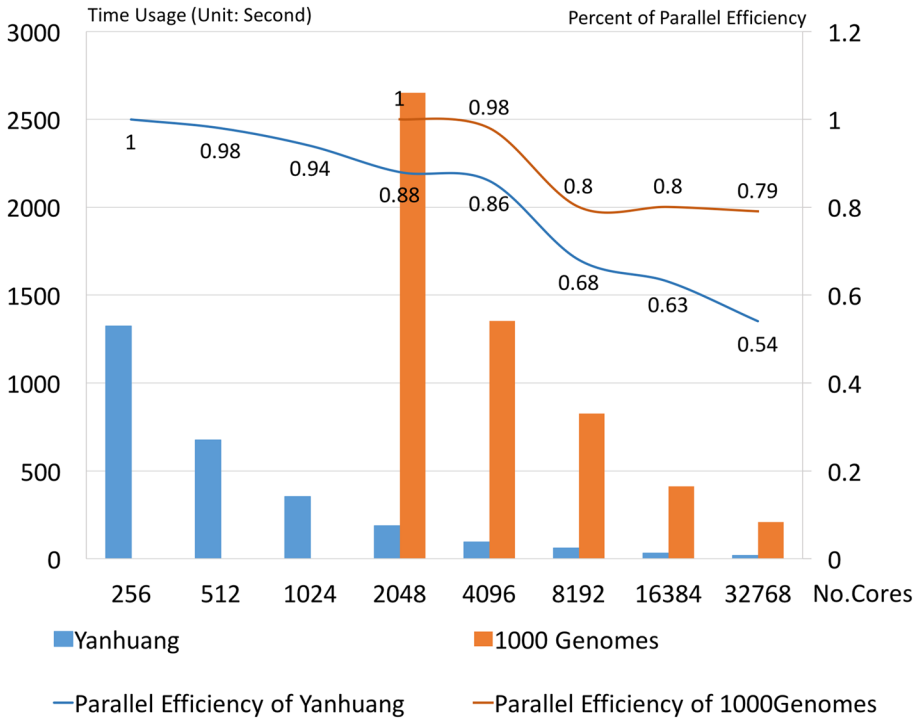

Fig. 6 Strong scaling test on Yanhuang and 1000 Genomes

2. *BFCounter* is the first one taking advantage of probabilistic data structure called bloom filter. Bloom filter allows a low rate of false positives while providing a very efficient memory management. Before inserting a kmer into the hash table, bloom filter can detect whether the current kmer has been observed or not. Thus, this technique saves about 50% memory consumption by discarding large quantities of singleton kmers, but it also greatly increases the whole processing time.
3. *MSPKmerCounter* is a disk-based approach, and it takes a novel strategy called Minimum Substring Partitioning (MSP) which breaks short-sequence reads into multiple disjoint partitions. Each partition in MSP can be loaded into memory and processed individually. Using minimizer [30] and super-kmers, the memory consumption greatly decreased. Besides, this technique also achieves astonishing compression ratio, so that it can reduced large amount of I/O operation. By taking advantage of this strategy, in many cases, MSPKmerCounter is faster than Jellyfish [15].
4. *KMC2* takes a novel method which optimizes the original minimizers into signature, which selects subset of all minimizers carefully and uses (k,x)-mers to speed up the whole process. The core difference between minimizer and signature is that signature tries to maintain a balance between size of each partition bin, which is a key factor that influences the overall performance. However, it should be noted that once the sequence data reach up to TB range, the distribution of minimizers will be uneven again, and this means that some partition bins will be extremely large. Finally, the time consumption does not proportionally increase as data increases.
5. *KHmer* entirely relies on a simple probabilistic data structure called Count-Min Sketch [31, 31]. This approach is similar to bloom filter. Count-Min Sketch uses multiple tables to detect whether current kmers appear more than once or not. Besides, KHmer only stores the count instead of storing kmers. Compared with other tools based on bloom filter, the key drawback of KHmer is that it just reserves the spectrum of kmers' frequency while lack the information of kmers and counting precision.

As the size of sequencing data increases dramatically in recent years, share memory tools have failed to handle these data beyond Terabytes. To accelerate this procedure and break the memory limit, several distributed tools have been developed.

1. *Kmerind* is the first kmer counting and indexing library for distributed memory environments. It applied optimization with efficient SIMD instructions and data structures. Kmerind also uses Bulk Synchronous Parallel (BSP) model on communication and fast two-level distribute sparse hash tables to enable fast inserting and updating of kmers. However, its scalability is limited (3072 as the largest from previous published works).

2. *Bloomfish* is another tool which overcomes the limit of single node, and its main strategy is leveraging Jellyfish building on top of memory efficient MapReduce framework-Mimir [27]. Bloomfish inherits most of the functions in Jellyfish, codesigning the I/O module and optimizing the memory management of the Mimir framework. Thus, Bloomfish also inherits most of the drawbacks of Jellyfish especially pre-estimating the number of distinct kmers.
3. *HipMer* is a high-quality end-to-end genome assembler developed with Unified Parallel C (UPC) language [32]. It also includes a kmer analysis module with the scalability of 15,360 cores. By first identifying frequency kmers (i.e., heavy hitters) and treating them specially, this strategy achieves much improvement in counting efficiency. However, this optimization does not impact all kinds of genomic data. Besides, to improve the efficiency of memory usage, HipMer also adopts bloom filters to eliminate singleton kmers.

6 Conclusion

Kmer counting is fundamental and critical for bioinformatics. As the unprecedented increase of data size and advanced sequencing technology, designing a novel method to count the abundance of these data efficiently has become increasingly important.

In this paper, we proposed a highly scalable counting tool for sequencing data profiling. This method replied on an optimized streaming I/O module, data pooling techniques, counting bloom filter, etc., to improve the algorithm performance and efficiency. The experimental results confirm that on single node, SWAPCounter shows competitive performance with shared memory kmer counting tools, SWAPCounter also present the best scalability and used least time when using 256 threads for Hg14 data set. With 4 TB sequences randomly selected from 1000 human genome project, the test on Cetus shows that our method can scale up to 32,768 cores with a 79% parallel efficiency, that is the best scalability compared with previous distributed kmer counting tool on TB data set.

One interesting point in future work is how to compress the gene data to make memory more efficient. Another interesting possibility is to optimize the communication framework to be more scalable. Finally, we integrate this tool into our genome assembler software to compress its memory usage and also improve its scalability.

Acknowledgements This work is supported by the National Key Research and Development Program of China under Grant nos. 2016YFB0201305 and 2018YFB0204403; National Science Foundation of China under Grant nos. U1435215 and 61433012; the Shenzhen Basic Research Fund under Grant nos. JCYJ20160331190123578, JCYJ20170413093358429, and GGFW2017073114031767; Chinese Academy of Sciences Grant under no. 2019VBA0009. We would also like to thank the funding support by the Shenzhen Discipline Construction Project for Urban Computing and Data Intelligence, Youth Innovation Promotion Association, and CAS to Yanjie Wei.

References

1. Zou Q, Li X, Jiang W, Lin Z, Li G, Chen K (2014) Survey of mapreduce frame operation in bioinformatics. Brief Bioinform 15(4):637–647
2. Guo R, Zhao Y, Zou Q, Fang X, Peng S (2018) Bioinformatics applications on apache spark. GigaScience 7(8):giy098
3. Miller JR, Koren S, Sutton GG (2010) Assembly algorithms for next-generation sequencing data. Genomics 95(6):315–327
4. Pevzner PA, Tang H, Waterman MS (2001) An eulerian path approach to DNA fragment assembly. Proc Nat Acad Sci 98(17):9748–9753
5. Meng J, Wang B, Wei Y, Feng S, Balaji P (2014) Swap-assembler: scalable and efficient genome assembly towards thousands of cores. BMC Bioinform BioMed Central 15(9):S2
6. Meng J, Seo S, Balaji P, Wei Y, Wang B, Feng S (2016) Swap-assembler 2: optimization of de novo genome assembler at extreme scale. In: Parallel processing (ICPP), 2016 45th international conference on. IEEE, pp 195–204
7. Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJ, Birol I (2009) Abyss: a parallel assembler for short read sequence data. Genome Res 19(6):1117–1123
8. Kelley DR, Schatz MC, Salzberg SL (2010) Quake: quality-aware detection and correction of sequencing errors. Genome Biol 11(11):R116
9. Liu Y, Schröder J, Schmidt B (2012) Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. Bioinformatics 29(3):308–315
10. Sheikhizadeh S, De Ridder D (2015) Ace: accurate correction of errors using k-mer tries. Bioinformatics 31(19):3216–3218
11. Medvedev P, Scott E, Kakaradov B, Pevzner P (2011) Error correction of high-throughput sequencing datasets with non-uniform coverage. Bioinformatics 27(13):i137–i141
12. Kent WJ (2002) Blat-the blast-like alignment tool. Genome Res 12(4):656–664
13. Qin J, Li R, Raes J, Arumugam M, Burgdorf KS, Manichanh C, Nielsen T, Pons N, Levenez F, Yamada T et al (2010) A human gut microbial gene catalogue established by metagenomic sequencing. Nature 464(7285):59
14. Marçais G, Kingsford C (2011) A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics 27(6):764–770
15. Li Y et al (2015) Mspkmercounter: a fast and memory efficient approach for k-mer counting. arXiv:1505.06550 (arXiv preprint)
16. Li Y, Kamousi P, Han F, Yang S, Yan X, Suri S (2013) Memory efficient minimum substring partitioning. Very Large Data Bases 6(3):169–180
17. Melsted P, Pritchard JK (2011) Efficient counting of k-mers in dna sequences using a bloom filter. BMC Bioinform 12(1):333
18. Deorowicz S, Kokot M, Grabowski S, Debudaj-Grabysz A (2015) Kmc 2: fast and resource-frugal k-mer counting. Bioinformatics 31(10):1569–1576
19. Rizk G, Lavenier D, Chikhi R (2013) Dsk: k-mer counting with very low memory usage. Bioinformatics 29(5):652–653
20. Zhang Q, Pell J, Caninokoning R, Howe A, Brown CT (2014) These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. PLoS One 9:7

21. Roy RS, Bhattacharya D, Schliep A (2014) Turtle: identifying frequent k-mers with cache-efficient algorithms. Bioinformatics 30(14):1950–1957
22. Perez N, Gutierrez M, Vera N (2016) Computational performance assessment of k-mer counting algorithms. J Comput Biol 23(4):248–255
23. Pan T, Flick P, Jain C, Liu Y, Aluru S (2017) Kmerind: a flexible parallel library for k-mer indexing of biological sequences on distributed memory systems. IEEE/ACM Trans Comput B
24. Gao T, Guo Y, Wei Y, Wang B, Lu Y, Cicotti P, Balaji P, Taufer M (2017) Bloomfish: a highly scalable distributed k-mer counting framework. In: ICPADS IEEE international conference on parallel and distributed systems, IEEE. Shenzhen, China: IEEE. [Online]. http://www.futurenet.ac.cn/icpads2017/?program-Gid_33.html
25. Georganas E, Buluç A, Chapman J, Hofmeyr S, Aluru C, Egan R, Oliker L, Rokhsar D, Yelick K (2015) Hipmer: an extreme-scale de novo genome assembler. In: Proceedings of the international conference for high performance computing. ACM, networking, storage and analysis, p 14
26. Georganas E, Buluç A, Chapman J, Oliker L, Rokhsar D, Yelick K (2014) Parallel de bruijn graph construction and traversal for de novo genome assembly. In: Proceedings of the international conference for high performance computing, networking, storage and analysis. IEEE Press, pp 437–448
27. Gao T, Guo Y, Zhang B, Cicotti P, Lu Y, Balaji P, Taufer M (2017) Mimir: memory-efficient and scalable mapreduce for large supercomputing systems. In: Parallel and distributed processing symposium (IPDPS), IEEE international. IEEE 2017, pp 1098–1108
28. Blustein J, El-Maazawi A (2002) Bloom filters: a tutorial, analysis, and survey. Dalhousie University, Halifax, pp 1–31
29. http://llimllib.github.io/bloomfilter-tutorial/
30. Roberts M, Hayes W, Hunt BR, Mount SM, Yorke JA (2004) Reducing storage requirements for biological sequence comparison. Bioinformatics 20(18):3363–3369
31. Cormode G, Muthukrishnan S (2005) An improved data stream summary: the count-min sketch and its applications. J Algorithms 55(1):58–75
32. Unified parallel c. http://upc.lbl.gov/