High-Frequency K-mer Counting at Low Memory Footprint

Li Mocheng,[1] Liu Yang,[1] Xiao Nong,[1] and Chen Zhiguang[2]
[1]*Institute for Quantum Information & State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, Changsha, China*
[2]*School of Computer, Sun Yat-sen University, Guangzhou, China*
Email: chenzhg29@mail.sysu.edu.cn

Genomics data analysis requires efficient tools to address the vast amount of data generated by current next-generation sequencing technologies. K-mer counting works face difficulties in balancing high memory overhead with statistical precision. We designed a high-frequency k-mer statistical computation based on the Space Saving algorithm and a novel hash table structure, which reduces the memory overhead by 46% while ensuring high computational efficiency.

Introduction: Data analysis in genomics relies on efficient k-mer analysis algorithms. With the development of next-generation sequencing (NGS) technology, the size of datasets has expanded dramatically, resulting in memory-intensive. With the application of high-performance computing devices, macro-genomic research parallelization schemes are widely used for sequence indexing [1][2], sequence alignment [3][4] and genome assembly [5] [6]. However, the simple parallelization of traditional algorithms poses significant performance problems. Targeted optimization schemes for storing short read sequences in large-scale scenarios are lacking. Instead, traditional hash tables or sketches are used, with a large room for optimization.

Short-read sequences suffer from high error rates. K-mer is a data type split from short-read sequences based on sliding windows. The traditional hash table-based k-mer storage structure keeps a large number of low-frequency error k-mer precisely, which leads to a more severe memory shortage. The sketch-based storage structure can effectively filter out low-frequency k-mer, but it needs to be composed of two parts, filtering and hashing, increasing the computational overhead. The k-mer counters are divided into two categories: exact indexing based on hash tables and coarse indexing by filters.

Pan[7] combines the cache-friendly Robinhood algorithm to achieve fast and accurate k-mer counting. However, it does not filter low-frequency k-mers, resulting in unnecessary data taking up much memory space. BFCounter [8] is a counter based on Bloom filters. It uses the filter to discard low-frequency data and stores only k-mers with frequencies greater than a threshold into the hash table. Similar is SWAPCounter [9], which chooses to sketch instead of Bloom filters. SWAPCounter is the most advanced technique for applying Count-min Sketch to k-mer counting. It roughly counts low-frequency k-mers by Count-min Sketch, and those exceeding the threshold enter the hash table for accurate counting, effectively reducing the memory occupation of the hash table. However, it suffers from false positives and extra memory access overhead, resulting in inefficient item access.

We completed the insertion in 82 seconds for 174GB fastq files. The query for one million k-mers in a batch is 1 second. Compared to cutting-edge k-mer hash tables, we reduce memory footprint by 46% with no performance loss.

Hash Table Structure: Figure 1 shows the structure of a bucket hash table and its access method. The hash table reduces the computational load through the bucket structure. Each item of the hash table consists of a bucket with M slots, each of which is a $< key, value >$ pair.

When a k-mer needs to be inserted or queried, it is hashed $N (N = 3$ in figure 1) times to match the N corresponding buckets that make up the candidate space. Then all the $M \times N$ slots are traversed, and the insertion or query algorithm is executed. The selected N buckets are used as candidates for the k-mer to be accessed. The $M \times N$ slots are called candidate slots.

The hash function uses Murmurhash, which has fast and uniform hashing properties and works well to support k-mer hash tables. The bucket traversal is designed to be depth-first, which can better guarantee the cache hit rate.

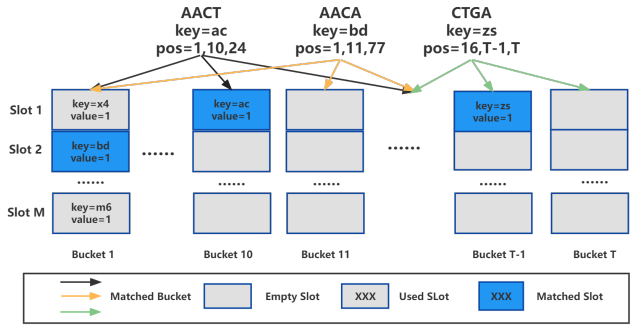

Fig 1 *Bucket Hash Table And Its Access Method*

Algorithm 1 Insertion

Require:
$kmer$ to be inserted
Number of Hashes M
Ensure:
Hash Table HT

```
1:  minPos ← (1, hash(kmer, 1));
2:  minEle ← HT[bucket[1]][1]
3:  key ← getKey(kmer);
4:  for i ← 1 to M do
5:      b ← hash(kmer, i)
6:      for j ← 1 to N do
7:          if HT[b][j].value == 0 then
8:              HT[b][j] ← (key, 1);
9:              Return;
10:         end if
11:         if HT[b][j].key == key then
12:             HT[b][j].value ← abs(HT[b][j].value) + 1;
13:             Return;
14:         end if
15:         if abs(minEle.value) > abs(HT[b][j].value) then
16:             minPos ← (b, j);
17:             minEle ← HT[b][j];
18:         end if
19:     end for
20: end for
21: HT[minPos].key ← Key;
22: HT[minPos].value ← −abs(minEle.value) − 1;
```

Insertion: The insertion algorithm is the core of this paper. We use the Space Saving algorithm in the insertion algorithm to solve the problem of full candidate buckets and allow the storage structure to be used properly with low memory overhead. As shown in Algorithm 1, we iterate through all the candidate slots corresponding to the k-mer through two levels and then find the empty slots to perform the insertion. When the slot is full, the Space Saving algorithm eliminates the k-mer with the lowest frequency and replaces it.

We have made specific improvements to make the filter more suitable for application to k-mer counting. In lines 21-22 of the algorithm, the new value is negative when the replacement strategy of Space Saving is triggered. Furthermore, in line 12, the recurring value is updated to a positive number. This improvement ensures that any positive number occurs at least twice and reduces the impact of low-frequency data.

Query: A k-mer query is an iteration of candidate slots as shown in algorithm 2. The main feature is identifying unique labels and avoiding contamination of high-frequency k-mer by low trust data. When the query result is less than 0(line 6), this k-mer is not trusted and returns 1. The rest of the cases work as regular hash tables.

Algorithm 2 Query

Require:
$kmer$ to be found
Ensure:
query result

```
key ← getKey(kmer);
for i ← 1 to M do
    b ← hash(kmer, i)
    for j ← 1 to N do
        if HT[b][j].key == key then
            if HT[b][j].value < 0 then
                return 1;
            else
                return HT[b][j].value;
            end if
        end if
    end for
end for
return 0;
```

Table 1. Experimental Datasets

Id	File Count	Size(GB)	Source	Accession
D1	1	2.9	NCBI	SRP072055
D2	11	13.8	NCBI	SRP004241
D3	3	54.3	NCBI	SRP003680
D4	3	174.1	NCBI	SRP003680

Results: Our selected datasets are shown in Table 1 with D1, D2, D3 and D4. Due to the large size of the data, we completed the experiments in the Tianhe-2 distributed parallel environment. One compute node on Tianhe-2 has 2 Intel Ivy Bridge CPUs, each with 12 cores and 32GB of memory, for 24 cores and 64GB of memory. We use up to 32 nodes for insert and query. We chose Robinhood hashing from Pan [7] and Count-min sketch from SWAPCounter [9] as the benchmark. Our goal is a smaller memory footprint than Robinhood hashing and faster data access than the sketch-based counter. We use SS as the short for our Space Saving-based counter, CM as the short for Count-min sketch-based counter, and RH as the short for Robinhood hashing. In the whole experiment, the k-mer has k = 31.

Table 2. Performance

Memory (% of D1)	Method	Frequency Deviation 1	$(1, 2^2]$	$(2^2, 2^4]$	$(2^4, +)$
130%	RH	0(133681)	0(143318)	0(143137)	0(579800)
130%	CM	-636	+384	+192	+60
130%	**SS**	**-74**	**-19**	**+2**	**0**
70%	RH	-28083	-50042	-56271	-232713
70%	CM	-4088	2522	+1491	+75
70%	**SS**	**-6084**	**-1176**	**+105**	**0**
35%	CM	-21014	+11787	+9054	+173
35%	**SS**	**-62174**	**-14499**	**+3836**	**-1406**

We take the memory occupation of the k-mer list extracted from D1 as 100% and compare the counting quality at 130%, 70%, and 35% memory occupation, which affects the number of slots in the storage structure. For example, the minimum memory footprint for D1 to run properly in RH is 2.6GB, so 130% memory footprint means 3.4GB of memory is used at all methods. The query results for the 1 million k-mers can be seen in Table.2. At 130% memory occupation, RH usually behaves, and we use it as the baseline for the following comparison. The number of k-kmers corresponding to the frequency is in the parentheses. At 70%, the RH data suffer from memory shortage and thus does not work, and the CM shows 0.2% false positive data. While SS maintained a false positive rate of less than 0.02%, the insert and query algorithm ensured that the frequency of these data must be greater than 2. At 35%, both CM and SS were plagued by insufficient memory, but the false positive rate was about 1% for CM and 0.3% for SS. In conclusion, if k-mers in $(4, +\infty)$ interval is required to be counted at the error rate less than 2‰, TopKmer's workable memory footprint is **54%** of Robinhood's and **70%** of SWAPCounter's.

Table 3. Order Impact

Memory Usage (% of D1)	Shuffle	Number of different k-mers in different frequencies 4	5	6	7	8	(8, +)
130%	S1	0	0	0	0	0	0
130%	S2	0	0	0	0	0	0
130%	S3	0	0	0	0	0	0
70%	S1	4	0	0	0	0	0
70%	S2	4	0	0	0	0	0
70%	S3	4	0	0	0	0	0
35%	S1	192	18	0	0	0	0
35%	S2	192	18	0	0	0	0
35%	S3	193	19	0	0	0	0

The order of the inputs influences the Space Saving algorithm's statistical results. We shuffle the dataset and analyze the content differences that appear under different insertion orders. Table 3 counts the number of different k-mers under different shuffles, and the comparison object is D1 without shuffling under 130% memory usage. Data discarding when conflict happens, and the discarded data are affected by the insertion order.

Our algorithm is little affected by the insertion order because most collisions and substitutions occur among low-frequency k-mers, and the high-frequency data statistics remain highly stable. The difference in insertion order will lead to the difference in the final statistics. However, the statistical accuracy can be guaranteed if the high-frequency k-mers do not switch frequently.

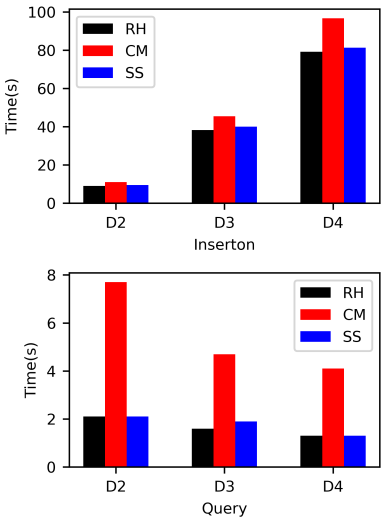

Fig 2 *Time Cost*

We compared the insertion and query time overheads at different datasets. As shown in figure 2, we achieved a similar time overhead with Robinhood hashing. Moreover, we achieved a lower time overhead than

SWAPCounter. We used 8, 16, and 32 nodes for computation on D2, D3, and D4 datasets, respectively, which resulted in different query times. On dataset D4, we completed the statistics using Robinhood hashing 49% of the memory with no performance degradation. We used the Count-min Sketch-based counter 85% of the time to complete insertions and 31% of the time to complete searches. We achieve an efficient memory access strategy and can outperform sketch-based structures and match state-of-the-art hash table-based structures.

Conclusion: The current k-mer counter suffers from the problem that computational efficiency and memory usage cannot be reconciled. To solve this problem, we designed a multi-layer hash table based on the Space Saving algorithm. We accomplish efficient data insertion and retrieval on a 174GB dataset. We reduce the memory overhead by 46% compared to the cutting-edge hash table structure and 30% compared to the cutting-edge sketch structure.

Acknowledgments: Thanks to the reviewers for their dedication to this paper. Thank you Guo Jiangyu, Gao Qianwen and Wang Yifeng for their encouragement and support.

NSFC: U1811461 the Major Program of Guangdong Basic and Applied Research: 2019B030302002 Supported by the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant NO. 2016ZT06D211. Guangdong Natural Science Foundation (2018B030312002)

Received: 10 January 2021 *Accepted:* 4 March 2021
doi: 10.1049/ell2.10001

References

1. Pan, T., et al.: Kmerind: A flexible parallel library for k-mer indexing of biological sequences on distributed memory systems. IEEE/ACM transactions on computational biology and bioinformatics 16(4), 1117–1131 (2017)
2. Fujimoto, M.S., Lyman, C.A., Clement, M.J.: Kcollections: A fast and efficient library for k-mers. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 193–198. IEEE (2020)
3. Georganas, E., et al.: meraligner: A fully parallel sequence aligner. In: 2015 IEEE International Parallel and Distributed Processing Symposium, pp. 561–570. IEEE (2015)
4. Chan, Y., et al.: Punas: A parallel ungapped-alignment-featured seed verification algorithm for next-generation sequencing read alignment. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 52–61. IEEE (2017)
5. Holley, G., Melsted, P.: Bifrost: highly parallel construction and indexing of colored and compacted de bruijn graphs. Genome biology 21(1), 1–20 (2020)
6. Ghosh, P., Krishnamoorthy, S., Kalyanaraman, A.: Pakman: A scalable algorithm for generating genomic contigs on distributed memory machines. IEEE Transactions on Parallel and Distributed Systems 32(5), 1191–1209 (2020)
7. Pan, T.C., Misra, S., Aluru, S.: Optimizing high performance distributed memory parallel hash tables for dna k-mer counting. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 135–147. IEEE (2018)
8. Melsted, P., Pritchard, J.K.: Efficient counting of k-mers in dna sequences using a bloom filter. BMC bioinformatics 12(1), 1–7 (2011)
9. Ge, J., et al.: Counting kmers for biological sequences at large scale. Interdisciplinary Sciences: Computational Life Sciences 12(1), 99–108 (2020)