Quill: A Memory Efficient k-mer Counting and k-mer Querying Tool for Commodity Clusters

Budvin Edippuliarachchi
University of Moratuwa
Sri Lanka
chathurae.17@cse.mrt.ac.lk

Damika Gamlath
University of Moratuwa
Sri Lanka
damikagamlath.17@cse.mrt.ac.lk

Ruchin Amaratunga
University of Moratuwa
Sri Lanka
ruchin.17@cse.mrt.ac.lk

Gunavaran Brihadiswaran
University of Moratuwa
Sri Lanka
gunavaran@cse.mrt.ac.lk

Sanath Jayasena
University of Moratuwa
Sri Lanka
sanath@cse.mrt.ac.lk

ABSTRACT

K-mer counting is a widely used fundamental bioinformatics process. With next generation sequencing and other advances in sequencing techniques, newly generated large sequence datasets demand efficient k-mer counting techniques that are capable of utilizing available resources. We present Quill: a memory-efficient k-mer counting and a querying tool for commodity clusters. While existing distributed memory solutions require high-performance clusters, Quill manages to perform k-mer counting in a conventional computer cluster without relying on high-performance network interfaces or parallel file systems. Furthermore, Quill provides an additional advantage in cases where k-mer counting is required for multiple k-values in the same dataset. Quill shows a linear scaling for the k-mer counting stage when tested in a commodity cluster. The performance gain is more evident when executed with k values up to 22 and 28. Thus, Quill can be viewed as a cost-effective k-mer counting solution that can effectively use the combined computing power of a cluster of commodity-grade computers. Quill is freely available at https://github.com/CSE-Optimizers/k-mer_counter.

CCS CONCEPTS

• **Applied computing** → **Bioinformatics**.

KEYWORDS

K-mer counting, Performance engineering, Parallel computing, Distributed computing

ACM Reference Format:
Budvin Edippuliarachchi, Damika Gamlath, Ruchin Amaratunga, Gunavaran Brihadiswaran, and Sanath Jayasena. 2022. Quill: A Memory Efficient k-mer Counting and k-mer Querying Tool for Commodity Clusters. In *2022 14th International Conference on Bioinformatics and Biomedical Technology (ICBBT 2022), May 27–29, 2022, Xi'an, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3543377.3543389

ICBBT 2022, May 27–29, 2022, Xi'an, China

ACM ISBN 978-1-4503-9638-7/22/05…$15.00
https://doi.org/10.1145/3543377.3543389

1 INTRODUCTION

K-mer counting is one of the fundamental tasks in bioinformatics. It is the process of counting the number of occurrences of k length substrings in a given sequence dataset. K-mer counting is important since it is a prerequisite for several bioinformatics processes including multiple sequence alignment[6], genome assembly[12], repeat detection[18], and sequence error correction[8, 14].

Although the k-mer counting procedure is a straightforward process, it has been an interesting research topic since k-mer counting is closely related to performance engineering with parallel computing. The large size of the datasets and the requirement of large amount of memory are two of the main challenges in k-mer counting process. Existing solutions are either optimized for high-performance clusters or medium to high end single machines. In this research, we present Quill, a k-mer counting and a k-mer querying tool for commodity clusters. The primary objective of Quill is to extract the combined computing power and the storage of a low-cost commodity cluster for k-mer counting rather than relying on high-performance hardware systems. Furthermore, Quill is designed to efficiently perform k-mer counting for multiple k values in the same dataset.

1.1 Related Work

Initially, most of the recognized solutions for k-mer counting were based on shared-memory systems. Jellyfish[16] can be considered as one of the first recognized k-mer counting tools. Jellyfish efficiently uses CAS instructions to implement a lock-free hash table for parallel k-mer insertion. KMC1[4] uses disk storage when the memory is not sufficient to store the k-mers for counting. This is implemented by categorizing k-mers into bins and dumping them onto the disk. KMC1 uses prefix-based mapping to assign k-mers into bins. KMC1 further overlaps I/O overhead and CPU time by using pipeline architecture. Later KMC2[5] which is the successor of KMC1, further reduces counting time and intermediate disk usage by using a concept called minimizers to categorize k-mers as a replacement for the prefix-based method in KMC1. KMC3[11], the successor of KMC2 further optimized KMC2. MSPKmerCounter[13] was another disk-based k-mer counter that used a method called Minimum Substring Partitioning (MSP) to map k-mers into partitions. MSP uses superkmers to map the k-mers into partitions without much data duplication for adjacent k-mers. MSP concept

also inspired the minimizers which are used in KMC2. In all the disk-based solutions, one of the goals was to design a good mapping strategy to evenly distribute the k-mers into partitions. Gerbil[7] is another disk-based k-mer counter that supports GPU acceleration. Gerbil also uses the minimizer concept and supports k-mer counting for larger k values (k>31). Although initial research was limited to shared memory solutions later solutions successfully utilized the combined memory capacity of clusters of computers to address high memory requirements. Kmerind[17] is a distributed memory-based solution designed for high-performance parallel k-mer counting and indexing. Bloomfish[9] is another highly scalable distributed k-mer counting framework for supercomputing clusters. Bloomfish is the successor of Jellyfish and is built on top of Mimir[10] which is an MPI based map-reduce framework.

It is evident from the literature that there is no existing efficient k-mer counting solution to effectively run on a commodity grade cluster. Existing distributed memory solutions depend on high-performance clusters with high-performance networks and parallel file systems. In this paper, we present Quill, a memory-efficient k-mer counting and a k-mer querying tool for commodity clusters. Quill supports DNA sequence datasets in FASTQ format and is capable of performing k-mer counting up to k=31. Additionally, Quill is further optimized for cases where k-mer counting is required for multiple k values in the same dataset. We designed a file compaction scheme that supports k-mer counting for multiple k values. With the optimizations, Quill is capable of using the combined computing power of a commodity cluster to achieve an overall better performance. Quill can be viewed as a cost-effective k-mer counting tool that is not dependent on high-end hardware systems. Quill is written in C++ and freely available at https://github.com/CSE-Optimizers/k-mer_counter under MIT license.

2 METHODOLOGY

Quill starts the workflow by reading the data file which is stored at the master node and sending file chunks to counting nodes. Counting nodes receive file chunks and store them in their local disks. After that, those chunks are reused to count the k-mers. K-mers are counted using a binning strategy with hashmap data structures. A separate tool is designed for k-mer querying purposes.

2.1 NFS Approach

Before coming up with the final solution, Quill was previously tested and experimented with various approaches. One approach was to store the data file in an NFS and counting nodes parallelly read that data file. With the NFS interface, the counting nodes access the data file with the same file reading procedures used for accessing local files. One cluster node was used as the NFS. In this approach, the data file was logically partitioned and those partitions were statically allocated to counting nodes. Counting nodes simultaneously access the data file. However, it was observed that this NFS file reading strategy was not an ideal solution in terms of scalability. We measured the time taken for reading the file and for the counting stage. Figure 1 elaborates the results of this experiment.

Although the counting time was reduced as expected, the increasing file reading time affected the overall time. This is because the

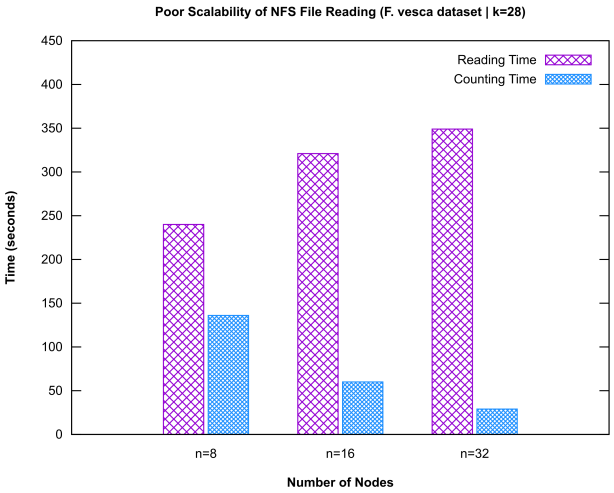

Figure 1: Poor scalability of NFS file reading

NFS does not provide truly parallel I/O. We further experimented with different read buffer sizes and also by changing the NFS daemon workers. However, no optimal strategy could be found. Hence it was decided to limit the file reading to one node.

2.2 Overview of the Solution

With the observations from the NFS approach, we designed the workflow of Quill with two main phases. The data file is stored at one node (referred to as the master node). A pipelined workflow is implemented where possible for the reading, processing, and writing stages.

- Phase 1 - contains 2 steps
  - Distribution - Master node reads from the data file and simultaneously distributes file chunks to counting nodes.
  - Compaction - Counting nodes receive file chunks, compact them, and store them in local disks
- Phase 2 - also contains 2 steps
  - Generation - Counting nodes reuse the previously created compact file chunks and generate k-mers. K-mers are mapped to bins and stored in the local disk.
  - Counting - Bins are loaded to memory and k-mers are counted using hashmap data structures.

The importance of designing with two phases is Quill can be efficiently used to perform k-mer counting for multiple k values. For this, phase 1 is needed to be executed only once and phase 2 can be repeated with different k values. This is achieved by reusing the compact version of the data file generated by phase 1. In the following sections, we provide a detailed description of each step.

2.3 File Distribution

In the file distribution step, the master node starts reading from the data file which is stored in its local disk. Specifically, the master node uses a 2 stage pipeline design, which is illustrated in Figure 2.

The reader thread fetches DNA reads from the file and forms chunks from them. A chunk will be a maximum of 256KB in size.

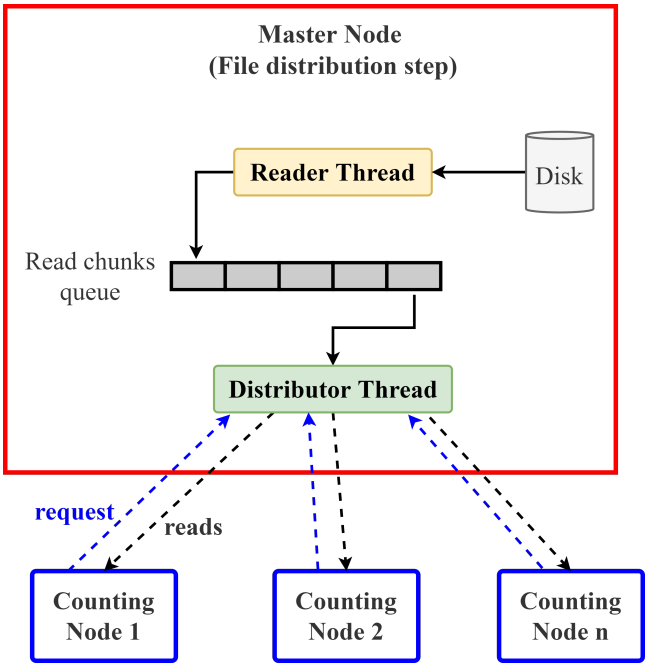

Figure 2: Pipeline at master node for file reading and distribution

Chunks are added to the read chunks queue. Upon receiving chunks, the distributor thread gets chunks one at a time and sends them to counting nodes. This communication is implemented with `MPI_Send` and `MPI_Recv` routines. Counting nodes continuously poll for new chunks. Upon receiving polling messages, the distributor thread sends chunks. This procedure works on a first-come-first-serve (FCFS) basis. Hence the chunks are not statically allocated. Load across the counting nodes is effectively balanced with the FCFS strategy. On the other hand, `MPI_Scatter` could have been another option to distribute the file chunks. We managed to implement it and test it as well. However, when `MPI_Scatter` is used, it results in more idle time in counting nodes which eventually increases the total distribution time. This is because rather than waiting to form a bigger chunk for distribution, it is efficient to distribute smaller chunks as soon as they are generated by the reader thread in the master node.

2.4 File Compaction

The motivation for using a compact data file is that it enables Quill to perform k-mer counting for multiple k values in the same dataset more efficiently. With the file compaction strategy, the distribution needs to be done only once when performing k-mer counting for multiple k values. This method reduces I/O overhead significantly, especially in commodity-grade clusters. We emphasize this idea in the results section.

The file compaction process is connected with the distribution pipeline. The counting nodes receive file chunks from the master node, compact the DNA reads, and finally write to the local disk. This is illustrated in Figure 3. The receiver thread handles communication with the master node. Recall that the counting nodes are continuously polling for new data chunks. The receiver thread executes that polling procedure. Received chunks are stored in the read chunks queue. The compaction thread pops chunks from this queue and encodes to a more compact version. Compacted chunks are sent to the compact chunks queue and the writer thread appends those chunks to a binary file in the local disk.

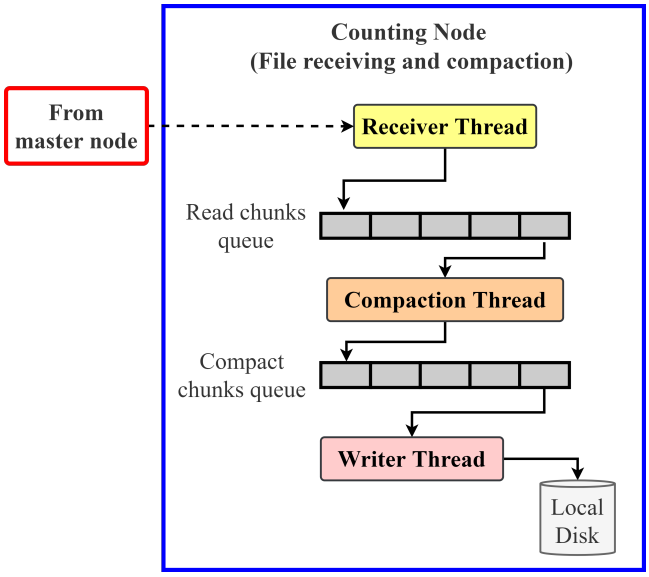

Figure 3: Pipeline at counting node for file chunk receiving and compaction

For the compaction method, we use a 2 step process. First, in the distribution phase, Quill only distributes the actual DNA reads from the FASTQ data file. With that, we remove the identifier data and quality score data. The second step is encoding the remaining DNA reads to a more compact version. For this, we use a 3-bit encoding and 32-bit 'blocks' to store 3-bit sets.

Quill is designed for FASTQ data files with a DNA alphabet which contains A, C, G and T as DNA bases and N as undefined or unidentified bases. We use the following 3-bit encoding for those characters.

A - 000 C - 001 G - 010 T - 011 N - 100

Since a block is 32 bits, 10 such encoded characters can be stored in a block. The remaining 2 bits were used as identifier bits. There are two types of blocks; header blocks and normal blocks. Normal blocks are the blocks containing the above-mentioned 3-bit encodings. Header blocks are placed at the start of each read. The header block contains the size of the read. The first 2 bits are used as identifiers to distinguish the two types of blocks. Header blocks contain 11 as the first 2 bits and the normal blocks contain 00 as the first 2 bits. With this method, if a DNA read has a length of l, the resulting encoding will have 1 header block and $\lceil l/10 \rceil$ number of normal blocks.

An example of this is provided in Figure 4. In that example, the length of the read is 83. The header block with identifier bits set

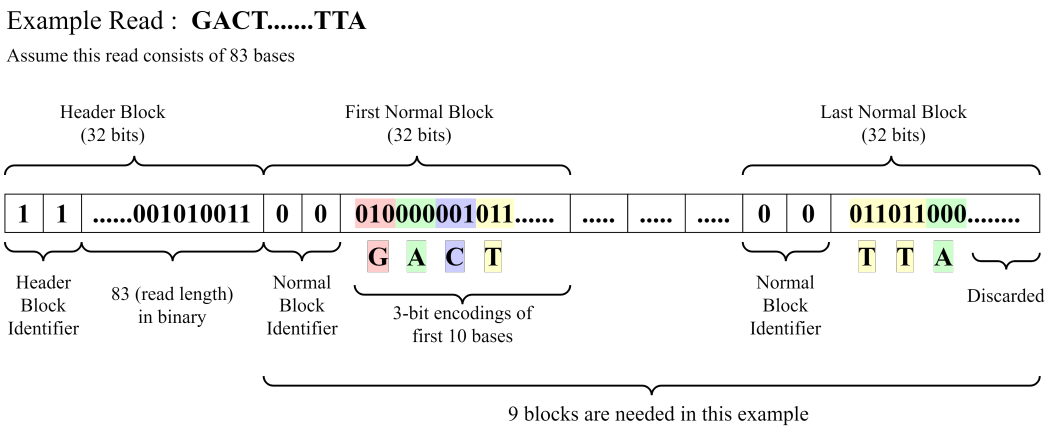

Figure 4: An example for the 3-bit encoding compaction scheme

to 11 contains the value 83 encoded in binary. Next 9 (= ⌈83/10⌉) blocks contain the 3-bit encoded characters.

The total compaction ratio can be calculated analytically. In the distribution stage, the overall size is reduced by at least 50% by eliminating identifiers and quality scores in the FASTQ file. In the 3-bit encoding stage, each byte in character is reduced to 3 bits, giving additional 37.5% compaction. The overall compaction is approximately 50% * 37.5% = 18.75%. However, because of using the header blocks, the actual value can be within the 17% - 18% range. With that strategy, we have been able to effectively eliminate 82% of the original size.

Phase 1 of Quill ends with the compaction step. After phase 1, each counting node consists of a compacted part of the original data file. With the available resources in the commodity-grade cluster, we were able to successfully pipeline phase 1 so that the complete distribution process completely overlapped with the file reading overhead. We emphasize this further in the results section.

As we mentioned before, Quill is designed to take advantage of situations where the k-mer counting is needed for multiple k values. To facilitate that, phase 1 can be executed once for distribution and phase 2 can be repeated for different k values. Phase 2 is described in the following sections.

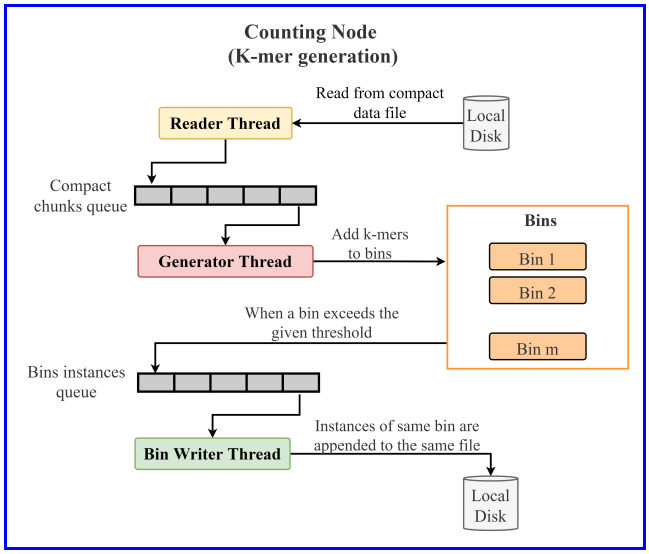

Figure 5: Pipeline at counting node for k-mer generation step

2.5 K-mer Generation

K-mer generation is the first step in phase 2. An overview of k-mer generation step is illustrated in Figure 5.

This is also a 3 stage pipeline with reading, processing, and writing steps. The reader thread reads from the previously generated compact data file. Chunks from the compact file are added to a queue. Then the generator thread generates k-mers from those chunks. In this step, the k-mers are also converted to the canonical version. A k-mer is encoded into a 32-bit buffer. Each character in a k-mer is encoded with 2 bits since Quill eliminates the undefined characters other than A, C, G and T for k-mer counting. For canonical k-mer generation, we use a similar method used in Frigate[3]. It is a moving window strategy to simultaneously generate k-mer and its reverse complement. For each incoming new character in the DNA read,

(1) To generate the k-mer
  (a) Left shift the current k-mer encoding by 2 bits
  (b) Clear the $(2k+1)^{th}$ bit and $(2k+2)^{th}$ bit. (assume 1 based bit indexing from LSB)
  (c) Perform a bitwise OR operation to place the 2-bit encoding of the incoming character in the 1^{st} bit and 2^{nd} bit positions
(2) To generate the k-mer reverse complement
  (a) Right shift the current complement k-mer encoding by 2 bits
  (b) Compute the complement character of the incoming character

(c) Perform a bitwise OR operation to place the 2-bit encoding of the computed complement character in the $(2k\text{-}1)^{th}$ bit and $2k^{th}$ bit positions

After that, each canonical k-mer is mapped to a bin using a mapper function. The mapper function is designed to distribute k-mers evenly across bins. We use the following mapper function to determine the corresponding bin for each k-mer. In this expression, *kmer* is a bit encoded 32 bit value and m is the number of bins.

$$bin\_id = (kmer >> (k/2)) \bmod m \quad (1)$$

With this mapper function we were able to achieve even distribution with canonical k-mers. Furthermore, we define a maximum size (s) of a bin. The values of m and s are determined according to the available memory in counting nodes. When a size of a bin exceeds s, it is added to the bin instances queue and a new empty bin is allocated to replace it. The bin writer thread gets bins from that queue and appends them to the corresponding binary file. For example, instances of the bin with id '2' are appended to the file with id '2'. After the k-mer generation step, each counting node contains m number of binary files with k-mers.

2.6 K-mer Counting

The actual k-mer counting step is the last step in phase 2 of Quill. K-mer counting step is started when the counting node has finished the k-mer generation step. K-mer counting step is also a 3 stage pipeline which is illustrated in Figure 6.

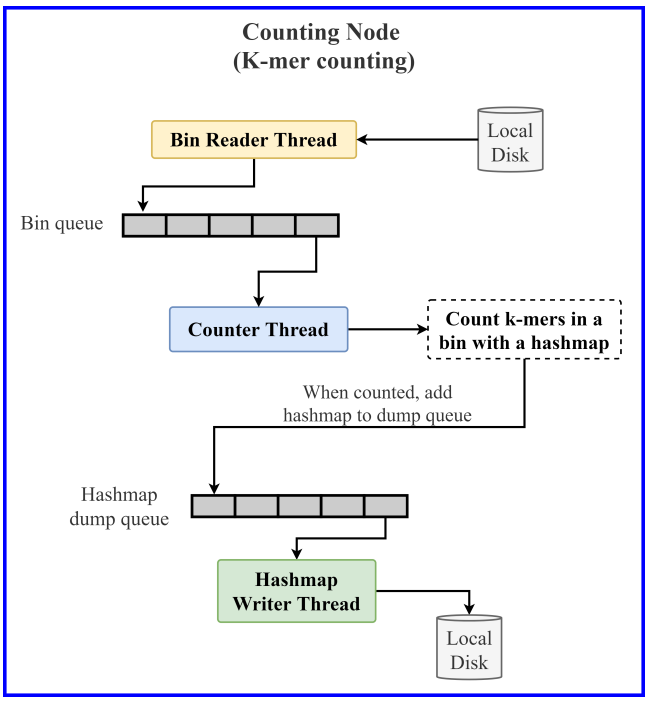

Figure 6: Pipeline at counting node for k-mer counting step

The bin reader thread loads previously created bins containing k-mers. Bins are added to the bin queue. Recall that, by using a mapper function each k-mer is uniquely mapped to a bin and no k-mer is present in multiple bins. Because of that k-mers inside bins can be counted separately. K-mers in a bin are counted using a hashmap data structure. Quill uses densehashmap[2] as the hashmap implementation. Furthermore, MurmurHash3[1] is used as the hash function. We used these configurations inspired by Kmerind. After a bin is finished with counting, the associated hashmap is sent to the dump queue and the hashmap writer thread serializes the hashmaps to files in the local disk. With this pipeline stage, Quill completes the whole k-mer counting process.

2.7 Querying Tool

Similar to KMC and other k-mer counters, Quill is designed with a separate tool to fetch the k-mer counting results. Quill is developed with more focus on k-mer querying similar to distributed querying in Kmerind. Additionally, Quill is capable of creating a final k-mer dump as well. For querying, the users can submit a set of k-mers to the master node. Then the master node distributes all the query k-mers to all the counting nodes using `MPI_Bcast` method. Upon receiving the query k-mers, the counting nodes group them according to the previously used mapper function. After that, the necessary hashmaps are identified to query the k-mers. In this phase, all the hashmaps need not be loaded into the memory. Only the hashmaps that contain the query k-mers are loaded into the memory and the counts are fetched locally in each counting node. After that counts are aggregated to the master node. For this, `MPI_Reduce` method is used with `MPI_SUM` operation. Aggregated results are returned to the user from the master node.

Apart from that, Quill is capable of generating a k-mer count dump. For this, hashmaps are merged at the master node. Specifically, all the hashmaps corresponding to one bin are merged as one batch. Then the next batch will be merged. Once a batch is finished merging, the resulting single hashmap is stored in the disk of the master node.

3 RESULTS

3.1 Experiment Setup

Since Quill targets commodity clusters consisting of low-medium grade computers, all the experiments were run on a cluster created using a university computer lab. All the computers were identical all-in-one PCs with Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz processor. The CPUs contain 2 physical cores and 2 threads per core. Each computer contains 4GB SODIMM DDR3 Synchronous 1600 MHz (0.6 ns) memory and the Hard Disk Drive is a 500GB TOSHIBA MQ01ABF050 with 5400 rpm. All the machines contain a conventional ext4 file system. NFS v4 was used in the earlier experiments where NFS was required to access the data file from all the counting nodes. The computers are connected with 1 Gbps ethernet without any specialized high-speed networking interfaces. Experiments were run on Ubuntu 14.04.5 LTS with MPICH v3.4.2 as the MPI implementation. One computer was used as the master node and all the experiments were initiated from that node. We executed Kmerind, KMC3 and Gerbil to compare with Quill. Kmerind was initiated from the same master node and that node was used to run KMC3 and Gerbil as well. Execution time was measured using the Linux time utility to get the elapsed wall clock time.

Table 1: Used Datasets

Dataset name	No. of bases (10^9)	FASTQ file size (GB)	Avg. read length	No. of reads
F. vesca	5.2	10.9	405	12,803,137
G. gallus	34.7	108	100	347,395,606
H. sapiens	123.7	292.1	151	819,148,264

3.2 Used Datasets

Referring to the related literature, especially from [15] we selected 3 datasets to evaluate Quill along with the existing k-mer counting tools. F. vesca, G. gallus and H. sapiens were used as the DNA sequence datasets. Table 1 contains all the details about the selected datasets.

Sources for the datasets can be found in the appendix section.

3.3 Execution Time of Quill

Quill was tested for k = 12, 15, 22, and 28 along with different numbers of cluster nodes. Fig 7 and 8 summarise the execution time for F. vesca and G. gallus datasets respectively. For any dataset, the time taken for phase 1, which is distribution and compaction, is almost the same regardless of the k value and the number of nodes. With a higher number of nodes, the phase 1 time slightly increased due to MPI startup time and network overhead but it is not significant. The reason for this behavior is Quill manages to completely overlap the distribution and compaction time with the file reading overhead. The complete phase 1 pipeline is fully utilized starting from data file reading to writing the compact data file. To confirm this we measured the time taken only to read the file inside the master node and it was nearly identical to the phase 1 time. On the other hand, this can be exploited if the users have a faster storage device at the master node. We only used a conventional 5400rpm HDD at the master node. If the end-users can use better storage (i.e. an SSD) only at the master node, the distribution time can be reduced significantly.

The time taken for phase 2, which is the k-mer generation and counting phase, is reducing as expected with a higher number of nodes. This confirms the efficient data parallelism of Quill. On some occasions, the scaling of phase 2 is better than linear scaling. We present 2 possible reasons for this superlinear behavior. One reason is, the percentage of the variance of execution time across the counting nodes gets reduced with a higher number of nodes. In other words, the counting time is more equally distributed with a higher number of nodes. This results in a better overall performance. The other reason is, with more counting nodes, the allocated data for a node is less and it can result in a low probability of hash collisions in the hashmap data structures. It also reduces the overall execution time.

3.4 K-mer Counting for Multiple K Values

Quill is designed to get the best performance advantage in cases where k-mer counting is needed for multiple k values in the same dataset. This was the primary motivation to store an intermediate compact version of the data file. We present the performance advantage of this aspect in Figure 9 and Figure 10. For a given dataset

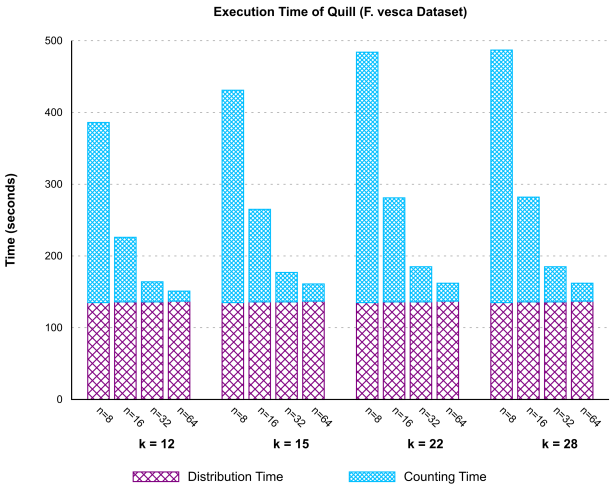

Figure 7: Execution time of Quill (F. vesca)

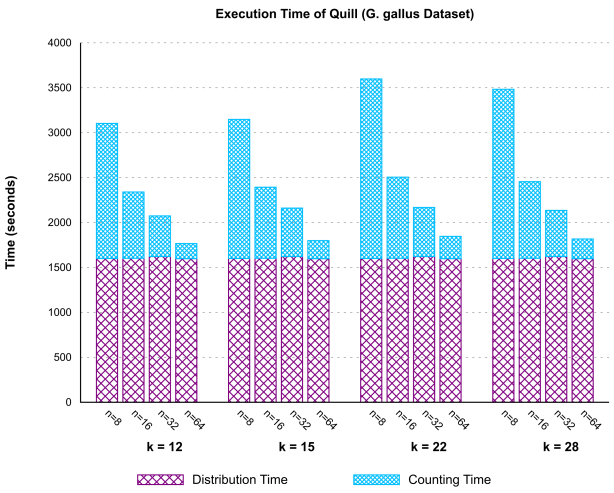

Figure 8: Execution time of Quill (G. gallus)

and a set of nodes, when performing k-mer counting for multiple k values, phase 1 is needed to be executed only once. It is denoted as the distribution time in the figures. Then phase 2, which is the counting phase, can be repeated with different k values by reusing the compact files generated by phase 1. Figure 9 and Figure 10 show the total time for k-mer counting for multiple k values. Time taken for phase 1 can be justified with the total execution time taken for k-mer counting for multiple k values. We can observe that total execution time shows a good scaling with the increasing number of nodes despite the distribution time being nearly equal. To the best of our knowledge in the related literature, Quill can be considered as an efficient solution in this particular use case.

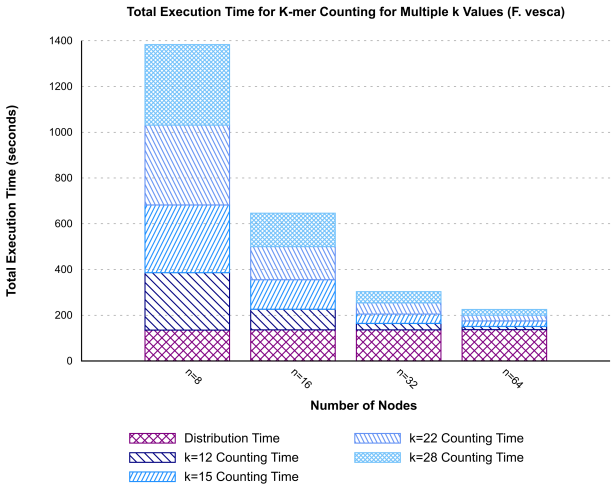

Figure 9: Execution time of Quill for multiple k values (F. vesca)

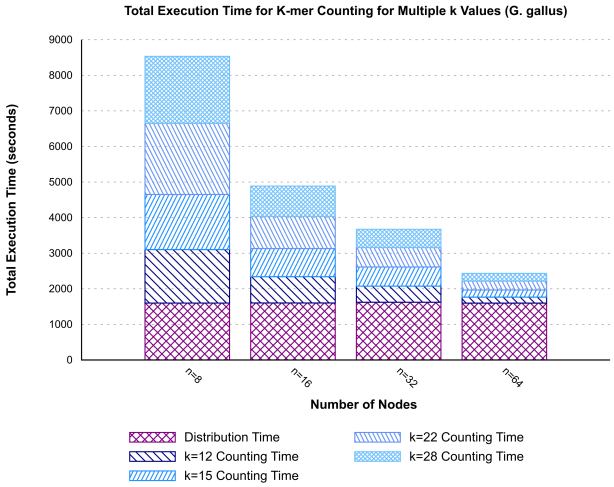

Figure 10: Execution time of Quill for multiple k values (G. gallus)

3.5 Comparison with Existing Soluions

Quill is directly comparable with existing k-mer counting solutions which are designed for distributed memory systems. In the literature, we identified Kmerind and Bloomfish as the current state of the art in that category. They both are capable of scaling efficiently with a larger number of cluster nodes. The main important fact about them is both Kmerind and Bloomfish are designed for high-performance clusters which are equipped with high-performance networking hardware and parallel file systems. However, we tested both of them in our commodity-grade cluster environment. In that experiment, Kmerind was not able to complete the process. Every time it stopped with memory assertion errors implying Kmerind needs a high amount of memory in counting nodes. Even for the smallest dataset (F. vesca) and with 64 nodes, Kmerind was not able to utilize the available memory to complete the process. On the other hand, Bloomfish was not compatible with its underlying map-reduce framework when compiling in our cluster environment. However, even if it was successful in compiling, Bloomfish can be negatively affected by the available hardware in the cluster. Bloomfish relies on parallel file reading from segments in the file. However, commodity-grade clusters do not possess such systems. Because of that, Bloomfish is not capable of running in its ideal parallel file reading pipeline with the available NFS settings in our cluster. Its performance can be significantly reduced due to the NFS file reading bottleneck we earlier mentioned. Because of these reasons and observations, Quill can be considered as one of a kind k-mer counting solution that is optimized for commodity-grade computer cluster systems. Quill is capable of extracting the combined computing power of such a cluster without depending on expensive high-performance hardware systems. In that aspect, Quill can be viewed as a cost-effective solution by considering the available hardware and the scalability performance.

Because of those issues with Kmerind and Bloomfish, we present a comparison with KMC3 and Gerbil, which are two of the state-of-the-art solutions developed for shared memory environments. We present these comparison results only to provide a benchmark for the hardware and software capabilities of a single computer. Here we emphasize the fact that while KMC3 and Gerbil are limited to resources in a single machine, Quill is capable of connecting the resources in low-grade computers to achieve an overall better performance. For this experiment, KMC3 and Gerbil were executed only in the master node. For both KMC3 and Gerbil we present the results when executed with 4 threads in the master node. Since the master node contains a 2 core CPU with 2 threads each, 4 thread configuration provided the best performance for KMC3 and Gerbil.

Table 2: Execution time(in seconds) of K-mer Counters For *F. vesca* Dataset

K-value	KMC3	Gerbil	Quill			
			n^*=8	n^*=16	n^*=32	n^*=64
12	170	546	386	226	164	151
15	351	487	431	265	177	161
22	304	482	484	281	185	162
28	291	473	487	282	185	162

n^* - number of nodes

Table 3: Execution time(in seconds) of K-mer Counters For *G. gallus* Dataset

K-value	KMC3	Gerbil	Quill			
			n^*=8	n^*=16	n^*=32	n^*=64
12	1729	3598	3103	2340	2074	1766
15	2558	3629	3148	2393	2161	1799
22	3353	3567	3598	2505	2168	1846
28	3141	3377	3483	2455	2135	1817

n^* - number of nodes

Results are summarized in the Tables 2 and 3. For both datasets, Quill is capable of achieving a lower execution time than KMC3 and Gerbil when provided with more computing nodes. The effect is more clear in the larger dataset, G gallus. The only exception is in the case where k=12 in KMC3. For such lower k values, KMC3 uses an in-memory algorithm which is obviously better than the disk-based approach. In all other cases, Quill manages to achieve a significantly better execution time with more nodes added to the cluster. To achieve a similar scaling behavior, KMC3 and Gerbil require a much better processor with a larger number of cores and also a much large memory in a single machine. However end-users do not always possess such systems but they might have access to a network of much lower grade computers, similar to a computer lab. Quill is designed specifically for such use cases. In that sense, we emphasize the fact that Quill is capable of combining a set of low-performance computers to achieve an overall better performance rather than relying on high-end hardware systems.

We also present these results viewed in another perspective where k-mer counting is required for multiple k values in the same dataset. Recall that phase 1 should be executed only once in such cases.

The advantage is much clear with the comparison in the Figure 11 and Figure 12. While KMC3 and Gerbil are required to restart the whole pipeline when counting for multiple k values, Quill is capable of reusing the compact data file which is generated in phase 1.

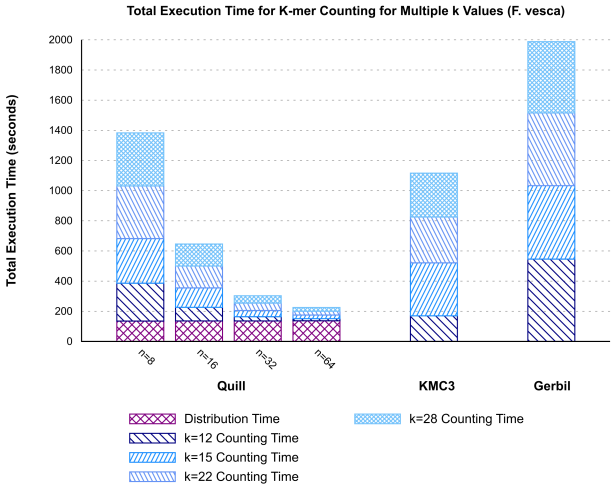

Figure 11: Comparison of Quill with KMC3 and Gerbil for multiple k values (F. vesca)

4 CONCLUSION

In this paper, we presented Quill; a memory-efficient k-mer counting and a k-mer querying tool for commodity cluster environments. Quill can be considered as a new strategy for utilizing hardware for k-mer counting since all the available k-mer counting tools for distributed memory systems are optimized for high-performance clusters only. Quill does not rely on specialized network interfaces

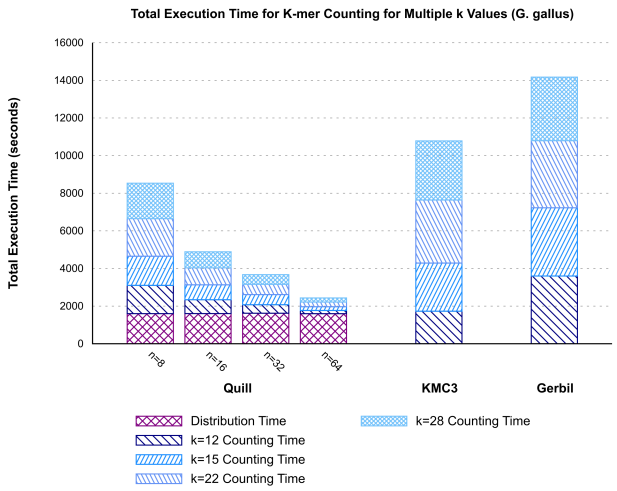

Figure 12: Comparison of Quill with KMC3 and Gerbil for multiple k values (G. gallus)

or parallel file systems. Because of that, Quill can be viewed as a more cost-effective solution that can be used in usual computer lab environments. Quill is capable of linear performance scaling for the counting stage with efficient data parallelism across cluster nodes. With a low memory of 4GB in each node, Quill is capable of utilizing the full pipeline in the distribution and counting stages. Furthermore, Quill is more suitable for use cases where k-mer counting is required for multiple k values in the same dataset. The compact data file which is generated using the compaction technique facilitates counting for multiple k-values without incurring repeated I/O overhead times. Currently, Quill supports k-mer counting up to k=31. We plan to improve Quill to support higher k values. Furthermore, we intend to improve the data distribution stage with a hierarchical design in the cluster environment.

ACKNOWLEDGMENTS

This research was supported by grant SRC/LT/2019/34 of the University of Moratuwa, Sri Lanka.

REFERENCES

[1] 2016. *MurmurHash3.* Retrieved April, 2022 from https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp
[2] 2020. *Google Sparsehash.* Retrieved April, 2022 from https://github.com/sparsehash/sparsehash
[3] Gunavaran Brihadiswaran and Sanath Jayasena. 2021. Frigate: A Fast, in-Memory Tool for Counting and Querying k-Mers. In *2021 13th International Conference on Bioinformatics and Biomedical Technology* (Xi'an, China) *(ICBBT 2021).* Association for Computing Machinery, New York, NY, USA, 134–140. https://doi.org/10.1145/3473258.3473279
[4] Sebastian Deorowicz, Agnieszka Debudaj-Grabysz, and Szymon Grabowski. 2013. Disk-based k-mer counting on a PC. *BMC bioinformatics* 14 (05 2013), 160. https://doi.org/10.1186/1471-2105-14-160
[5] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. 2015. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics* 31, 10 (01 2015), 1569–1576. https://doi.org/10.1093/bioinformatics/btv022 arXiv:https://academic.oup.com/bioinformatics/article-pdf/31/10/1569/17085507/btv022.pdf
[6] Robert C. Edgar. 2004. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research* 32, 5 (03 2004), 1792–1797.

https://doi.org/10.1093/nar/gkh340 arXiv:https://academic.oup.com/nar/article-pdf/32/5/1792/7055030/gkh340.pdf
[7] Marius Erbert, Steffen Rechner, and Matthias Müller-Hannemann. 2016. Gerbil: A Fast and Memory-Efficient k-mer Counter with GPU-Support. In *Algorithms in Bioinformatics*, Martin Frith and Christian Nørgaard Storm Pedersen (Eds.). Springer International Publishing, Cham, 150–161. https://doi.org/10.1007/978-3-319-43681-4_12
[8] M. Fujimoto, P. M. Bodily, N. Okuda, M. J. Clement, and Q. Snell. 2014. Effects of error-correction of heterozygous next-generation sequencing data. *BMC Bioinformatics* 15 Suppl 7 (2014), S3. https://doi.org/10.1186/1471-2105-15-S7-S3
[9] Tao Gao, Yanfei Guo, Yanjie Wei, Bingqiang Wang, Yutong Lu, Pietro Cicotti, Pavan Balaji, and Michela Taufer. 2017. Bloomfish: A Highly Scalable Distributed K-mer Counting Framework. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. 170–179. https://doi.org/10.1109/ICPADS.2017.00033
[10] Tao Gao, Yanfei Guo, Boyu Zhang, Pietro Cicotti, Yutong Lu, Pavan Balaji, and Michela Taufer. 2017. Mimir: Memory-Efficient and Scalable MapReduce for Large Supercomputing Systems. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1098–1108. https://doi.org/10.1109/IPDPS.2017.31
[11] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. 2017. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics* 33, 17 (05 2017), 2759–2761. https://doi.org/10.1093/bioinformatics/btx304 arXiv:https://academic.oup.com/bioinformatics/article-pdf/33/17/2759/25163903/btx304.pdf
[12] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, S. Li, H. Yang, J. Wang, and J. Wang. 2010. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res* 20, 2 (Feb 2010), 265–272. https://doi.org/10.1101/gr.097261.109
[13] Yang Li and Xifeng Yan. 2015. MSPKmerCounter: A Fast and Memory Efficient Approach for K-mer Counting. *ArXiv* abs/1505.06550 (2015).
[14] Yongchao Liu, Jan Schröder, and Bertil Schmidt. 2012. Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data. *Bioinformatics* 29, 3 (11 2012), 308–315. https://doi.org/10.1093/bioinformatics/bts690 arXiv:https://academic.oup.com/bioinformatics/article-pdf/29/3/308/17103359/bts690.pdf
[15] Swati Manekar and Shailesh Sathe. 2018. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience* 7 (10 2018). https://doi.org/10.1093/gigascience/giy125
[16] G. Marçais and C. Kingsford. 2011. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics* 27, 6 (Mar 2011), 764–770. https://doi.org/10.1093/bioinformatics/btr011
[17] Tony Pan, Patrick Flick, Chirag Jain, Yongchao Liu, and Srinivas Aluru. 2016. Kmerind: A Flexible Parallel Library for K-mer Indexing of Biological Sequences on Distributed Memory Systems. https://doi.org/10.1145/2975167.2975211
[18] A. L. Price, N. C. Jones, and P. A. Pevzner. 2005. De novo identification of repeat families in large genomes. *Bioinformatics* 21 Suppl 1 (Jun 2005), i351–358. https://doi.org/10.1093/bioinformatics/bti1018

A APPENDIX

A.1 Datasets

The datasets were downloaded from the following sources.

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

G. gallus
https://www.ebi.ac.uk/ena/data/view/SRX043656
(SRR105788, SRR105789, SRR105792, SRR105794, SRR197985, SRR197986)

H. sapiens
https://dnanexus-rnd.s3.amazonaws.com/NA12878-xten/reads/NA12-878D_HiSeqX_R1.fastq.gz
https://dnanexus-rnd.s3.amazonaws.com/NA12878-xten/reads/NA12-878D_HiSeqX_R2.fastq.gz

A.2 Experimental Results of Quill

Table 4: Execution time for all the experiments

Dataset	k	n*	Phase 1 time (seconds)	Phase 2 time (seconds)	Total time (seconds)
F. vesca	12	8	135	251	386
F. vesca	12	16	136	90	226
F. vesca	12	32	136	28	164
F. vesca	12	64	137	14	151
F. vesca	15	8	135	296	431
F. vesca	15	16	136	129	265
F. vesca	15	32	136	41	177
F. vesca	15	64	137	24	161
F. vesca	22	8	135	349	484
F. vesca	22	16	136	145	281
F. vesca	22	32	136	49	185
F. vesca	22	64	137	25	162
F. vesca	28	8	135	352	487
F. vesca	28	16	136	146	282
F. vesca	28	32	136	49	185
F. vesca	28	64	137	25	162
G. gallus	12	8	1600	1503	3103
G. gallus	12	16	1602	738	2340
G. gallus	12	32	1622	452	2074
G. gallus	12	64	1598	168	1766
G. gallus	15	8	1600	1548	3148
G. gallus	15	16	1602	791	2393
G. gallus	15	32	1622	539	2161
G. gallus	15	64	1598	201	1799
G. gallus	22	8	1600	1998	3598
G. gallus	22	16	1602	903	2505
G. gallus	22	32	1622	546	2168
G. gallus	22	64	1598	248	1846
G. gallus	28	8	1600	1883	3483
G. gallus	28	16	1602	853	2455
G. gallus	28	32	1622	513	2135
G. gallus	28	64	1598	219	1817
H. sapiens	12	32	2993	1331	4324
H. sapiens	12	64	3008	787	3795
H. sapiens	15	32	2993	1598	4591
H. sapiens	15	64	3008	837	3845
H. sapiens	22	32	2993	1906	4899
H. sapiens	22	64	3008	930	3938
H. sapiens	28	32	2993	1697	4690
H. sapiens	28	64	3008	934	3942

n* - number of nodes

A.3 Commands Used for Other Tools

- KMC3
  time ./kmc -k12 -m4 vesca.fastq ves_12.res tmp/
  time ./kmc -k15 -m4 vesca.fastq ves_15.res tmp/
  time ./kmc -k22 -m4 vesca.fastq ves_22.res tmp/
  time ./kmc -k28 -m4 vesca.fastq ves_28.res tmp/
  time ./kmc -k12 -m4 gallus.fastq ves_12.res tmp/
  time ./kmc -k15 -m4 gallus.fastq ves_15.res tmp/
  time ./kmc -k22 -m4 gallus.fastq ves_22.res tmp/
  time ./kmc -k28 -m4 gallus.fastq ves_28.res tmp/
  time ./kmc -k12 -m4 sapiens.fastq ves_12.res tmp/
  time ./kmc -k15 -m4 sapiens.fastq ves_15.res tmp/
  time ./kmc -k22 -m4 sapiens.fastq ves_22.res tmp/
  time ./kmc -k28 -m4 sapiens.fastq ves_28.res tmp/

- Gerbil
  time ./gerbil -k 12 -e 4GB -t 4 -l 2 vesca.fastq tmp/ out
  time ./gerbil -k 15 -e 4GB -t 4 -l 2 vesca.fastq tmp/ out
  time ./gerbil -k 22 -e 4GB -t 4 -l 2 vesca.fastq tmp/ out
  time ./gerbil -k 28 -e 4GB -t 4 -l 2 vesca.fastq tmp/ out
  time ./gerbil -k 12 -e 4GB -t 4 -l 2 gallus.fastq tmp/ out
  time ./gerbil -k 15 -e 4GB -t 4 -l 2 gallus.fastq tmp/ out
  time ./gerbil -k 22 -e 4GB -t 4 -l 2 gallus.fastq tmp/ out
  time ./gerbil -k 28 -e 4GB -t 4 -l 2 gallus.fastq tmp/ out
  time ./gerbil -k 12 -e 4GB -t 4 -l 2 sapiens.fastq tmp/ out
  time ./gerbil -k 15 -e 4GB -t 4 -l 2 sapiens.fastq tmp/ out
  time ./gerbil -k 22 -e 4GB -t 4 -l 2 sapiens.fastq tmp/ out
  time ./gerbil -k 28 -e 4GB -t 4 -l 2 sapiens.fastq tmp/ out

- Kmerind
  Used the example driver code provided in the repository.
  https://github.com/ParBLiSS/kmerind/blob/master/test/benchmark/BenchmarkKmerIndex.cpp