KMC 2: Fast and resource-frugal k-mer counting

Article *in* Bioinformatics · July 2014

Impact Factor: 4.98 · DOI: 10.1093/bioinformatics/btv022 · Source: arXiv

4 authors, including:

Marek Kokot
Silesian University of Technology
1 PUBLICATION **10** CITATIONS
SEE PROFILE

Szymon Grabowski
Lodz University of Technology
105 PUBLICATIONS **578** CITATIONS
SEE PROFILE

BIOINFORMATICS

Vol. 00 no. 00 2014
Pages 1–21

KMC 2: Fast and resource-frugal k-mer counting

Sebastian Deorowicz [1]*, Marek Kokot [1], Szymon Grabowski [2], Agnieszka Debudaj-Grabysz [1]

[1]Institute of Informatics, Silesian University of Technology, Akademicka 16, 44-100 Gliwice, Poland
[2]Computer Engineering Department, Technical University of Łódź, Al. Politechniki 11, 90-924 Łódź, Poland

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Associate Editor: XXXXXXX

ABSTRACT

Motivation: Building the histogram of occurrences of every k-symbol long substring of nucleotide data is a standard step in many bioinformatics applications, known under the name of k-mer counting. Its applications include developing de Bruijn graph genome assemblers, fast multiple sequence alignment and repeat detection. The tremendous amounts of NGS data require fast algorithms for k-mer counting, preferably using moderate amounts of memory.

Results: We present a novel method for k-mer counting, on large datasets at least twice faster than the strongest competitors (Jellyfish 2, KMC 1), using about 12 GB (or less) of RAM memory. Our disk-based method bears some resemblance to MSPKmerCounter, yet replacing the original minimizers with signatures (a carefully selected subset of all minimizers) and using (k, x)-mers allows to significantly reduce the I/O, and a highly parallel overall architecture allows to achieve unprecedented processing speeds. For example, KMC 2 allows to count the 28-mers of a human reads collection with 44-fold coverage (106 GB of compressed size) in about 20 minutes, on a 6-core Intel i7 PC with an SSD.

Availability: KMC 2 is freely available at http://sun.aei.polsl.pl/kmc.

Contact: sebastian.deorowicz@polsl.pl

1 INTRODUCTION

One of common preliminary steps in many bioinformatics algorithms is the procedure of *k-mer counting*. This primitive consists in counting the frequencies of all k-long strings in the given collection of sequencing reads, where k is usually more than 20, and has applications in de novo assembly using de Bruijn graphs, correcting reads and repeat detection, to name a few areas. More applications can be found, e.g., in (Marçais and Kingsford, 2011), with references therein.

K-mer counting is arguably one of the simplest (both conceptually and programmatically) tasks in computational biology, *if we do not care about efficiency*. The number of existing papers on this problem suggests however that efficient execution of this task, with reasonable memory use, is far from trivial. The most successful of early approaches was Jellyfish (Marçais and Kingsford, 2011), maintaining a compact hash table (HT) and using lock-free operations to allow parallel updates. The original Jellyfish version (as presented in (Marçais and Kingsford, 2011)) required more than 100 GB of memory to handle human genome data with 30-fold coverage. BFCounter (Melsted and Pritchard, 2011) employs the classic compact data structure, Bloom filter (BF), to reduce the memory requirements due to preventing most single-occurrence k-mers (which are usually results of sequencing errors and for most applications can be discarded) from being added to a hash table. Although BF is a probabilistic mechanism, BFCounter applies it in a smart way, which does not produce counting errors. Unfortunately, BFCounter is single-threaded and its performance is not competitive (see also the experimental results in (Deorowicz *et al.*, 2013)). DSK (Rizk *et al.*, 2013) and KMC (Deorowicz *et al.*, 2013) are two disk-based algorithms. On a high level, they are similar and partition the set of k-mers into disk buckets, which are then separately processed. DSK is more memory frugal and may process human genome data in as little as 4 GB of RAM, while KMC is faster but typically uses about 11–16 GB of RAM. Turtle (Roy *et al.*, 2014) bears some similarities to BFCounter. The standard Bloom filter is there replaced with its cache-friendly variant (Putze *et al.*, 2009) and the hash table is replaced with a sorting and compaction algorithm (which, accidentally, resembles a component of KMC), apart from adding parallelism and a few smaller modifications. Finally, MSPKmerCounter (Li and Yan, 2014) is another disk-based algorithm, based on the concept of minimizers, described in detail in the next section.

In this paper we present a new version of KMC, one of the fastest and most memory efficient programs. The new release borrows from the efficient architecture of KMC 1 but reduces the disk usage several times (sometimes about 10 times) and improves the speed usually about twice. In consequence, our tests show that KMC 2 is the fastest (by a far margin) algorithm for counting k-mers, with even smaller memory consumption than its predecessor.

There are two main ideas behind these improvements. The first is the use of signatures of k-mers that are a generalization of the idea of *minimizers* (Roberts *et al.*, 2004a,b). Signatures allow significant reduction of temporary disk space. The *minimizers* were used for the first time for the k-mer counting in MSPKmerCounter, but our modification significantly reduces the main memory requirements (up to 3–5 times) as well as disk space (about 5 times) as compared to MSPKmerCounter. The second main novelty is the use of (k, x)-mers ($x > 0$) for reduction of the amount of data to sort. Simply speaking, instead of sorting some amount of k-mers we sort a much

*to whom correspondence should be addressed

smaller portion of $(k + x)$-mers and then obtain the statistics for k-mers in the postprocessing phase.

2 METHODS

2.1 Minimizers of k-mers

Most k-mer counting algorithms start in the same way: they process each read from left to right and extract all k-mers from them, one by one. Although the destination for k-mers (hash table in Jellyfish, Bloom filter in BFCounter, disk in DSK and KMC 1) and other details differ in particular solutions, the first step remains essentially the same. There is high redundancy in such approach as consecutive k-mers share $k - 1$ symbols.

An obvious idea of reducing the redundancy is to store (in some way) a number of consecutive k-mers (ideally even a complete read) in one place. Unfortunately, to collect the statistics we need to find all copies of each unique k-mer, which is not an easy task when the copies are stored in many places. A clever solution to these problems is based on the concept of minimizers (Roberts *et al.*, 2004a,b). A *minimizer* of a k-mer is such of its m-mers ($m < k$) that no other lexicographically smaller m-mer can be found. The crucial observation is that usually many consecutive k-mers have the same minimizer, so in memory or in a file on disk they can be represented as one sequence of more than k symbols, significantly reducing the redundancy.

The idea of minimizers was adopted recently for k-mer counting (Li and Yan, 2014). Since in genomic data the read direction is rarely known, k-mer counters usually do not distinguish between direct k-mers and their reverse complements, and collect statistics for *canonical* k-mers. The canonical k-mer is lexicographically smaller of the pair: the k-mer and its reverse complement. Therefore, Li and Yan in their MSPKmerCounter use *canonical minimizers*, i.e., the minima of all canonical m-mers from the k-mer. They process the reads one by one and look for contiguous areas containing k-mers having the same canonical minimizer; they dub these areas as "super k-mers". Then, the resulting super k-mers are distributed into one of several *bins* (disk files) according to the related canonical minimizer (more precisely, according to its hash value; in this way the number of resulting bins is kept within reasonable limits). In the second stage each bin is loaded into main memory (one by one), all k-mers are extracted from the super k-mers, and then counted using a hash table; after processing a bin the entries from the hash table are dumped to disk and the hash table memory reclaimed. Since each bin contains only a small fraction of all k-mers present in the input data, the amount of memory necessary to process the bin is much smaller that in the case of whole input data.

This elegant idea allows to significantly reduce the disk space compared to storing each k-mer separately (as KMC 1 and DSK do). Unfortunately, it has the following drawbacks:

1. The distribution of bin sizes is far from uniform. In particular, the bin associated with the minimizer AA...A is usually huge. Other minimizers with a few As in their prefix also tend to produce large bins.
2. When a minimizer starts with a few As, then it often implies several new super k-mers spanning a single k-mer only. To given an example, with $m = 7$ and AAAAAAC as the minimizer: when the minimizer falls off the sliding window, so the current k-mer starts with AAAAAC, then AAAAACX (for some X) will likely be the new minimizer; but unfortunately for yet another window AAAACXY (for some Y) also has a fair chance to be a minimizer, etc.

As the amount of main memory needed by MSPKmerCounter is directly related to the number of k-mers in the largest bin, especially the former issue is important. It will be shown in the experimental section that the file corresponding to the minimizer AA...A can be really large.

2.2 From minimizers to signatures

To overcome the aforementioned problems we resign from "pure" minimizers and prefer to use the term of *signatures* of k-mers. Essentially, a signature can be any m-mer of k-mer, but in this paper we are interested in such signatures that solve both of the problems mentioned above. Namely, good signatures of length m should satisfy the following conditions:

1. The size of the largest bin should be as small as possible.
2. The number of bins should be neither too large nor too small.
3. The sum of bin sizes should be as small as possible.

Point 1 is obvious as it limits the maximum amount of needed memory. Point 2 protects from costly operations on a large number of files (open, close, append, etc.) in case of too many bins but also from load balancing difficulties on a multi-core system when the number of bins is small. The last point refers to the disk space, so minimizing it reduces the total I/O.

Obtaining optimal signatures, i.e., such that cannot be improved in any of the listed aspects, seems hard, so a compromise must be found. Since the origin of both problems are runs of As (especially as signature prefixes), we propose to use as signatures canonical minimizers, but only such that do not start with AAA, neither start with ACA, neither contain AA anywhere except at their beginning. We note that in earlier works on minimizers (Roberts *et al.*, 2004a,b; Wood and Salzberg, 2014) similar problems were spotted (in different applications) and somewhat different solutions were presented.

As the experiments show (cf. experimental section of the paper), such a modification significantly reduces the size of the largest bin and also reduces the total number of super k-mers, therefore both the main memory and temporary disk use is much smaller compared to using just canonical minimizers.

2.3 (k, x)-mers

In the memory-frugal k-mer counters (DSK, KMC 1, MSPKmerCounter) all the input k-mers are split into parts to reduce the amount of RAM memory necessary to store all the k-mers in explicit form. Then, the k-mers are sorted, inserted into a hash table or Bloom filter. Nevertheless, often the size of the largest part (bin) can be a problem, i.e., affects the peak RAM use. Also, there is a need to explicitly process (sort, insert into some data structure) each single k-mer.

Below we show that it is possible to reduce the amount of memory necessary for collecting the statistics even more and also speed up the sorting process by processing a significant part of k-mers implicitly. To this end, we need to introduce (k, x)-mers that are $(k + x')$-mers in the canonical form, where $x' = 0, 1, \ldots, x$ (for some small x) such that all k-mers within (k, x)-mer are in canonical form.

The idea is that instead of breaking super k-mers into k-mers (for sorting purposes), we break them into as few as possible (k, x)-mers in such way that no two neighbors share the same k-mer, but each k-mer present in a super k-mer is present in some of (k, x)-mers. As preliminary experiments on real data show, with setting $x = 3$ the number of (k, x)-mers becomes about twice smaller than the number of k-mers. This means that the main memory is reduced almost twice. At the same time, the sorting speed is improved.

2.4 Sketch of the algorithm

Similarly to its predecessor, KMC 2 has two phases: distribution and sorting. In the distribution phase, the reads are read from FASTQ/FASTA files. Each read is scanned to find (partially overlapping) regions (super k-mers) sharing the same signature (Fig. 1). These super k-mers are sent to bins (disk files) related to signatures. The number of possible signatures, 4^m, can be, however, quite large, e.g., 16,384 for typical value $m = 7$. Thus, to reduce the number of bins to at most 512, some signatures are merged (i.e., the corresponding sequences are sent to the same bin). To decide which

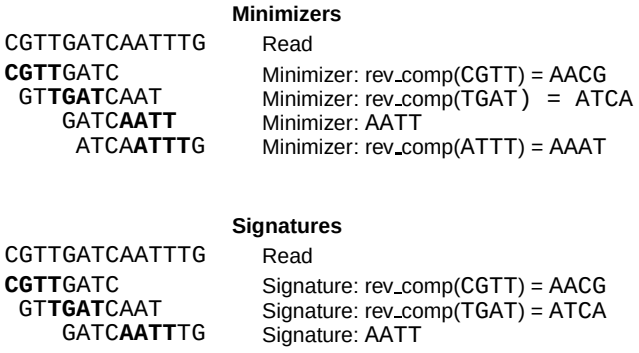

Fig. 1. A toy example of splitting a read into super k-mers. The assumed parameters are: $k = 8$, $m = 4$.

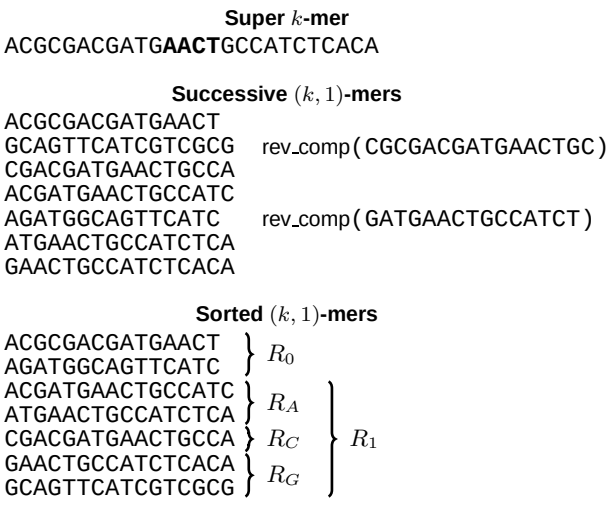

Fig. 2. Splitting a super k-mer into $(k,1)$-mers followed by sorting them. The assumed parameters are: $k = 15$, $m = 4$. The range R_T is empty (thus not shown).

signatures to merge, in a preprocessing stage KMC 2 reads a small fraction of the input data, builds a histogram of found signatures, and finally merges the least frequent signatures.

In the sorting phase, KMC 2 reads a file, extracts the (k,x)-mers from super k-mers and performs radix sort algorithm on them. Then, it calculates the statistics for k-mers. In real implementation x can be 0, 1, 2, or 3, but for presentation clarity we will describe how to collect the statistics of k-mers from $(k,1)$-mers.

It is important to notice where in the sorted array of $(k,1)$-mers some k-mer can be found. There are 6 possibilities:

1 it can be just a k-mer,

2 it can be a prefix of some $(k+1)$-mer,

3–6 it can be a suffix of $(k+1)$-mer preceded by A, C, G, or T.

Therefore, we conceptually split the array of $(k,1)$-mers into 5 non-overlapping, sorted subarrays: one (R_0) containing k-mers and four (R_A, R_C, R_G, R_T) containing $(k+1)$-mers starting with A, C, G, T. There is also one extra subarray (R_1) containing all $(k+1)$-mers, i.e., a concatenation of R_A, R_C, R_G and R_T (Fig. 2).

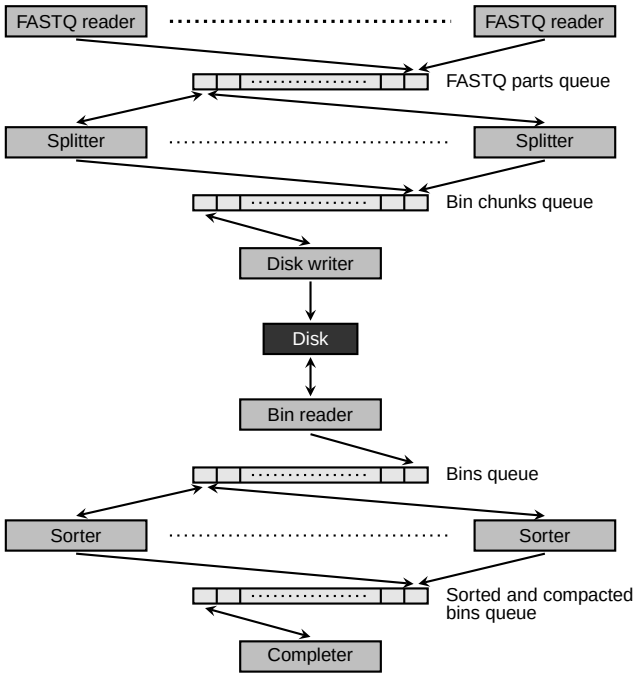

Fig. 3. A scheme of the parallel KMC algorithm

Now to collect the statistics of k-mers we scan these 6 subarrays in parallel. So, we have 6 pointers somewhere in R_* We compare the pointed elements, find the lexicographically smallest canonical k-mer among them (from R_X for X being a letter we take the suffix of $(k+1)$-mer) and store it in the resulting array of statistics of k-mers P if it is different than the recently added k-mer to P. Otherwise, we just increase the counter related to this k-mer in P. Since, we scan the arrays R_* in a linear fashion, the time complexity of this "merging" subphase is linear.

The overall KMC 2 algorithm is presented in Fig. 3. Several FASTQ readers send input data chunks into a queue, handled then by splitters which dispatch super k-mers with the same signature to the same bin chunk. The queue of these chunks is in turn processed with a disk writer, which dumps the bin to disk. In the next phase, the bins, read from disk to a queue in the memory, are sorted and compacted by multiple sorter threads. Finally, the completer stores the sorted bins in the output database on disk.

The final database of k-mers is stored in compact binary form. The KMC 2 package contains: the k-mer counter, dump program that allows to produce the textual list of k-mers together with their counters, C++ API designed to allow to use the database directly in various applications. The k-mer counter allows to specify various parameters, e.g., the threshold below which the k-mer is discarded (e.g., in some applications the k-mers appearing only once are treated as erroneous), the maximal amount of memory used in the processing. More details on the API, the database format and the search algorithm in the database are given in the Supplementary material.

2.5 Additional features

KMC 2, like its former version, allows to refrain from counting too rare or too frequent k-mers. It is done during "merging" substage, in which the total number of occurrences of each k-mer is known. The software also supports quality-aware counters, compatible with the popular error-correction package Quake (Kelley *et al.*, 2010). In this mode, the counter for the k-mer is incremented by the probability that all symbols of the k-mer are correct (calculated according to the base quality values). To allow this,

Table 1. Characteristics of the datasets used in the experiments.

Organism	Genome length	No. bases	FASTQ file size	No. files	Gzipped size	Avg. read length
F. vesca	210	4.5	10.3	11	3.5	353
G. gallus	1,040	34.7	115.9	15	25.9	100
M. balbisiana	472	56.9	197.1	2	49.1	101
H. sapiens 1	3,093	86.0	223.3	6	70.8	100
H. sapiens 2	3,093	135.3	312.9	48	105.8	101

No of bases are in Gbases. File sizes are in Gbytes (1Gbyte = 10^9 bytes). Approximate genome lengths are in Mbases according to http://www.ncbi.nlm.nih.gov/genome/.

the qualities must be stored in temporary disk files for each base of a super k-mer. To our knowledge, the only other k-mer counters with this functionality are KMC 1 and Jellyfish 1 (but not the current version 2). KMC 2 handles not only sequencing reads (FASTQ), but also genomes (FASTA). Finally, we note that KMC 2 can work in RAM-only mode in which the bins are simply stored in the main memory, which may be convenient for large datacenters.

3 RESULTS

The implementation of KMC 2 was compared against the best, in terms of speed and memory efficiency, competitors: Jellyfish 2 (which is significantly more efficient than the version described in (Marçais and Kingsford, 2011)), DSK, Turtle, MSPCounter, KAnalyze and KMC 1. Each program was tested for two values of k (28 and 55) and in two hardware configurations: using conventional disks (HDD) and using a solid-state disk (SSD). We used several datasets (Table 1) of varying size; two of them are human data with large coverage. The experiments were run on a machine equipped with an Intel i7 4930 CPU (6 cores clocked at 3.4 GHz), 64 GB RAM, and 2 HDDs (3 TB each) in RAID 0 and single SSD (1 TB). The programs were run with the number of threads equal to the number of virtual cores ($6 \times 2 = 12$), to achieve maximum speed.

The comparison, presented in Tables 2–4 and Supplementary Tables 1–2, includes total computation time (in seconds), maximum RAM use, maximum disk use. RAM and disk use are given in GBs (1 GB = 2^{30} B). Time is wall-clock time in seconds. A test running longer than 10 hours was interrupted. Other reasons for not finishing a test were excessive memory consumption (limited by the total RAM, i.e., 64 GB) or excessive disk use (over 650 GB, chosen for our 1 TB SSD disk; note that the largest input dataset, *H. sapiens* 2, occupies 312.9 GB on the same disk).

Several conclusions can easily be drawn from the presented tables. Two of the competitors, KAnalyze and MSPKC, are clearly the slowest; for this reason, KAnalyze was tested only on the SSD. KAnalyze also uses a large amount of temporary disk space, which was the reason we stopped its execution on the two human datasets (for $k = 28$ only, as KAnalyze does not support large values of k). MSPKC, on the other hand, theoretically allows the parameter k to exceed 32, but in none of our datasets it finished its work for $k = 55$; for the smallest dataset (*F. vesca*) it failed probably because of variable-length reads, on the other datasets we stopped it after more than 10 hours of processing. The only asset of KAnalyze and MSPKC we have found is their moderate memory use.

DSK is not very fast either. Still, it consistently uses the smallest amount of memory (6 GB was always reported) and is quite robust, as it passed all the tests.

Jellyfish 2 is not very frugal in memory use, and this is the reason on our machine it passed the test for $k = 55$ only for two datasets (*F. vesca* and *M. balbisiana*). Still, for $k = 28$ it passed all the tests, being one of the fastest programs, often outperforming KMC 1.

Table 2. k-mers counting results for *G. gallus*.

Algorithm	$k = 28$ RAM	$k = 28$ Disk	$k = 28$ Time	$k = 55$ RAM	$k = 55$ Disk	$k = 55$ Time
			SSD			
Jellyfish 2	33	0	880	*out of memory*		
KAnalyze	9	270	11,071	*unsupported k*		
DSK	6	101	1,325	6	94	1,836
Turtle	48	0	1,004	*out of memory*		
MSPKC	17	114	3,382	*out of time ($>$ 10 hours)*		
KMC 1	13	101	868	12	173	1,792
KMC 2 (12GB)	12	25	408	12	18	503
KMC 2 (6GB)	6	25	431	6	18	562
			HDD			
Jellyfish 2	33	0	915	*out of memory*		
DSK	6	101	3,600	6	94	4,206
Turtle	48	0	1,058	*out of memory*		
MSPKC	17	114	4,853	*out of time ($>$ 10 hours)*		
KMC 1	11	101	1,320	12	173	2,036
KMC 2	12	25	587	12	18	656

Table 3. k-mers counting results for *M. balbisiana*.

Algorithm	$k = 28$ RAM	$k = 28$ Disk	$k = 28$ Time	$k = 55$ RAM	$k = 55$ Disk	$k = 55$ Time
			SSD			
Jellyfish 2	17	0	1,080	26	0	853
KAnalyze	9	354	8,249	—	—	—
DSK	6	164	2,356	6	138	2,962
Turtle	46	0	1,484	*out of memory*		
MSPKC	10	185	8,729	*out of time ($>$ 10 hours)*		
KMC 1	13	165	1,229	15	279	2,622
KMC 2 (12GB)	12	41	755	12	29	834
KMC 2 (6GB)	6	41	685	6	29	895
			HDD			
Jellyfish 2	17	0	1,115	26	0	881
DSK	6	164	6,216	6	138	7,228
Turtle	46	0	1,498	*out of memory*		
MSPKC	10	185	12,152	*out of time ($>$ 10 hours)*		
KMC 1	13	165	2,194	15	279	3,367
KMC 2	12	41	960	12	29	1,041

Turtle is rather fast as well (slower than Jellyfish though), but even more memory hungry; we could not have run it on the two largest datasets. Turtle and Jellyfish are memory-only algorithms, all the other ones are disk-based. This is the reason why changing HDD to a much faster SSD does not affect the performance of these two counters significantly (yet it is non-zero due to faster input reading from the SSD).

KMC 2 on the SSD was tested twice for each k: with standard memory use (12 GB) and with reduced memory use (6 GB). These settings are a "suggestion" rather than a rigid limitation, as a large maximum bin size may force KMC 2 to use more memory, Such a phenomenon was seen several times especially in the memory-reduced runs. This also means that our goal

Table 4. k-mers counting results for *H. sapiens* 2.

Algorithm	$k=28$ RAM	$k=28$ Disk	$k=28$ Time	$k=55$ RAM	$k=55$ Disk	$k=55$ Time
SSD						
Jellyfish 2	62	0	3,212	*out of memory*		
KAnalyze	*out of disk (> 650 GB)*			*unsupported k*		
DSK	6	263	5,487	6	256	7,732
Turtle	*out of memory*			*out of memory*		
MSPKC	*out of time (> 10 hours)*			*out of time (> 10 hours)*		
KMC 1	17	396	2,998	*out of disk (> 650 GB)*		
KMC 2 (12GB)	12	101	1,615	13	70	2,038
KMC 2 (6GB)	6	101	1,706	13	70	2,446
HDD						
Jellyfish 2	62	0	3,231	*out of memory*		
DSK	6	263	18,493	6	256	22,432
KMC 1	17	396	4,898	*out of disk (> 650 GB)*		
KMC 2	12	101	2,259	13	70	2,640

Table 5. Influence of input data format on the k-mers counting times of KMC 2 for *H. sapiens* 2.

Algorithm	$k=28$ RAM	$k=28$ Disk	$k=28$ Time	$k=55$ RAM	$k=55$ Disk	$k=55$ Time
Non-gzipped input files						
KMC 2^{HDD}	12	101	2,259	13	70	2,640
KMC 2^{SSD}	12	101	1,615	13	70	2,038
KMC 2^{SSD}	6	101	1,706	13	70	2,446
Gzipped input files						
KMC 2^{HDD}	12	101	2,004	13	70	2,495
KMC 2^{SSD}	12	101	1,217	13	70	1,607
KMC 2^{SSD}	7	101	1,495	13	70	1,909

Table 6. Comparison of signatures and minimizers for *G. gallus* dataset.

Length	Minimizers: Avg. in read	Minimizers: No. k-mers largest bin	Minimizers: Min. memory	Signatures: Avg. in read	Signatures: No. k-mers largest bin	Signatures: Min. memory
			$k=28$			
5	6.935	3,361	26.5	6.045	1,904	18.1
6	7.519	1,231	10.9	6.385	625	5.9
7	7.919	641	5.5	6.728	283	2.6
8	8.304	371	3.1	7.143	328	3.0
			$k=55$			
5	2.669	3,940	62.0	2.477	2,257	38.3
6	2.915	1,513	24.7	2.591	819	13.9
7	3.038	801	12.8	2.642	280	5.5
8	3.117	467	7.3	2.678	330	6.4

'Avg. in read' is the average no. of super k-mers per read. 'No. k-mers largest bin' is the number (in millions) of k-mers in the largest bin. 'Min. memory' is the amount of memory (in Gbytes) necessary to process the k-mers in the largest bin, i.e., the lower bound of the memory requirements. The size of temporary disk space is determined by the average number of minimizers/signatures in a read. For example, the disk space requirements for minimizer/signature length 7 are: 25.4 GB (signatures, $k = 28$), 28.6 GB (minimizers, $k = 28$).

to match DSK in memory use in the memory-reduced mode was not quite accomplished, yet we note that reducing the memory resulted in processing time longer by only 5%–20%.

KMC 2 with its standard memory use is a clear winner in processing time, on the human datasets being about twice faster than Jellyfish 2 or KMC 1. These speed differences concern the SSD experiments, as on the HDD the gap diminishes (but is still significant). This can be explained by I/O (especially reading the input data) being the bottleneck in several phases of KMC 2 processing.

It is worth examining how switching a conventional disk to a SSD affects the performance of disk-based software. It might seem natural that the biggest time reduction (in absolute time, not percentage gain) should be seen in those programs which use more disk space. To some degree it is true (e.g., KMC 1 gains more than KMC 2) but DSK is a "counter-example": e.g., on *H. sapiens* 2 it gains a whopping 13,006 s which is almost seven times the reduction for KMC 1, seemingly surprising as DSK uses less disk space. Yet, a probable explanation is that DSK works in several passes, so its total I/O is actually quite large for large datasets.

Interestingly, for disk-based algorithms the disk use of KMC 2 is typically reduced when switching from $k = 28$ to $k = 55$. This can be explained by a smaller number of k-mers per read, and in case of KMC 2 also by a smaller number of super k-mers per read.

We also measured how the input format (raw, gzipped) and media (HDD, SSD) affects the performance of our solution on the largest dataset, *H. sapiens* 2 (Table 5). As expected, using the SSD reduces the time by 25%–40%, and reading the input from compressed form also has a visible positive impact. We note in passing that replacing gzip with, e.g., bzip2 (results not shown here) would not be a wise choice, since the improvement in compression cannot offset much slower bzip2's decompression.

Table 6 compares signatures with minimizers on *G. gallus*. We can see that using our signatures diminishes the average number of super k-mers in a read by about 10–15 percent. Also the number of k-mers in the largest (disk) bin is significantly reduced, sometimes more than twice. These achievements directly translate to smaller RAM and disk space consumption.

How (k, x)-mers affect bin processing is shown in Table 7 for two datasets. It is easy to see that the number of strings to sort is more than halved for $x = 3$, yet the speedup is more moderate, due to the extra split phase and sorting over longer strings. Still, $(k, 3)$-mers vs. plain k-mers reduce the total time by more than 20% (and even 38% for *H. sapiens* 2 and $k = 55$).

The impact of k on processing time and disk space is presented in Figures 4 and 5, respectively. Longer k-mers result in even longer super k-mers, which minimizes I/O, but makes the sorting phase longer. For this reason, the disk space consumption shrinks smoothly with growing k (Fig. 5), but the effect on processing time (Fig. 4) is not so clear. Still, counting k-mers for $k \geq 32$ is generally slower than for smaller values of k.

From Fig. 6 we can see that using more memory accelerates KMC 2, but the effect is mediocre (only about 10% speedup when raising the memory consumption from 16 GB to 40 GB). The reasons behind the speedup are basically 2-fold: (i) the extra RAM allows to use a larger number of sorter threads (which is more efficient than few sorters with more internal threads per sorter), and (ii) occasional large bins disallow to run other sorters at the same time if memory is limited.

Table 7. Impact of (k, x)-mers on bin processing and overall KMC 2 processing, for *G. gallus* and *H. sapiens* 2. 12 GB RAM set, gzipped input. "Sorted fraction" is the ratio of the number of (k, x)-mers to the number of k-mers.

x	$k = 28$ Split time	Sort time	Total time	Sorted fraction	$k = 55$ Split time	Sort time	Total time	Sorted fraction
				G. gallus				
0	102	159	261	1.000	98	381	479	1.000
1	127	131	258	0.646	104	284	388	0.639
2	127	119	246	0.539	104	265	369	0.527
3	127	112	239	0.491	106	240	346	0.479
				***H. sapiens* ERA015743**				
0	672	867	1539	1.000	399	2188	2587	1.000
1	664	669	1333	0.648	448	1480	1928	0.638
2	644	614	1258	0.541	455	1176	1630	0.526
3	644	573	1217	0.495	439	1168	1607	0.478

For *H. sapiens* 2 the largest bin was too large to fit the assumed amount of RAM in two cases, and the RAM consumption of KMC 2 was 25 GB for $(55, 0)$-mers, 18 GB for $(55, 1)$-mers, 15 GB for $(55, 2)$-mers, and 13 GB for $(55, 3)$-mers.

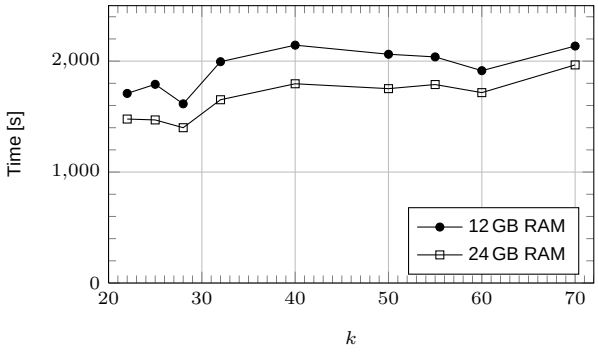

Fig. 4. Dependence of KMC 2 processing time on k for *H. sapiens* 2 dataset ($k = 22, 25, 28, 32, 40, 50, 60, 70$)

Finally, we analyze the scalability and CPU load of our software (Fig. 7). As expected, the highest speed is achieved when the number of threads matches the number of (virtual) CPU cores (12). Still, the time reduction between 1 and 12 threads is only by factor 3 or less, when the input data are in non-compressed FASTQ. Using the compressed input broadens the gap to factor 6.4 for $k = 28$ and 4.9 for $k = 55$. The corresponding gaps between 1 and 6 threads (i.e., equal to the number of *physical* cores) are: 2.3 and 2.5 ($k = 28$ and $k = 55$) with non-compressed input, and 4.9 and 3.9 ($k = 28$ and $k = 55$) with gzipped input. The latter experiment tells more about the scalability of our tool, since the performance boost from Intel hyper-threading technology can be hard to predict, varying from less than 10% (Schuepbach *et al.*, 2013, Tab. 1) to about 60% (Sebastião *et al.*, 2012, Tab. II) in real code.

4 CONCLUSION

Although the dominating trend in IT solutions nowadays is the cloud, the progress in bioinformatic algorithms shows that even home computers, equipped with multi-core CPUs, several gigabytes of RAM and a few fast hard disks (or one SSD disk) get powerful enough to be applied for real "omics" tasks, if their resources are loaded appropriately.

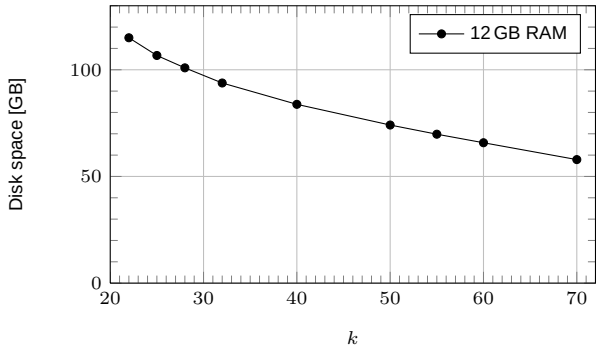

Fig. 5. Dependence of KMC 2 temporary disk usage on k for *H. sapiens* 2 dataset

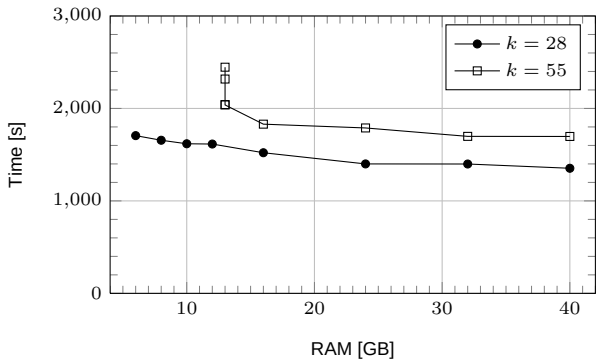

Fig. 6. Dependence of KMC 2 processing time on maximal available RAM and type of disk for *H. sapiens* 2 dataset. There are 4 results for $k = 55$ and 13 GB RAM. These results are for set 6 GB, 8 GB, 10 GB, 12 GB as maximal RAM usage. However, the largest bin enforced to spend at least 13 GB of RAM

The presented KMC 2 algorithm is currently the fastest k-mer counter, with modest resource (memory and disk) requirements. Although the used approach is similar to the one from MSPKmerCounter, we obtain an order of magnitude faster processing, due to the following KMC features: replacing the original minimizers with signatures (a carefully selected subset of all minimizers), using (k, x)-mers and a highly parallel overall architecture. As opposed to most competitors, KMC 2 worked stably across a large range of datasets and test settings.

In real numbers, we show that it is possible to count the 28-mers of a human reads collection with 44-fold coverage (106 GB of compressed size) in about 20 minutes, on a 6-core Intel Core i7 PC with an SSD. With enough amounts of available RAM it is also possible to run KMC 2 in memory only. In our preliminary tests it almost did not help compared to an SSD (up to 5% speedup) but may be an option in datacenters, with plenty of RAM but possibly using network HDDs with relatively low transfer. In this scenario a memory-only mode should be attractive.

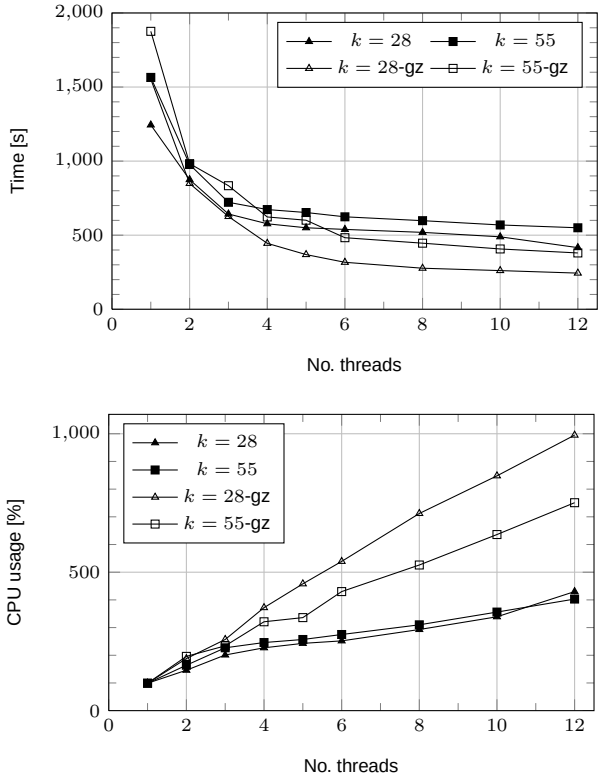

Fig. 7. Dependence of KMC 2 processing time and CPU usage on the set number of threads for *G. gallus* dataset.

We expect to successfully apply KMC 2 for a few problems related to k-mers, e.g., finding nullomers (Falda *et al.*, 2014).

ACKNOWLEDGMENT

Funding: This work was supported by the Polish National Science Centre under the project DEC-2012/05/B/ST6/03148. The work was performed using the infrastructure supported by POIG.02.03.01-24-099/13 grant: "GeCONiI–Upper Silesian Center for Computational Science and Engineering".

REFERENCES

Deorowicz, S., Debudaj-Grabysz, A., and Grabowski, S. (2013). Disk-based k-mer counting on a PC. *BMC Bioinformatics*, **14**, 160.

Falda, M., Fontana, P., Barzon, L., Toppo, S., and Lavezzo, E. (2014). keeSeek: searching distant non-existing words in genomes for PCR-based applications. *Bioinformatics*, doi:10.1093/bioinformatics/btu312.

Kelley, D. R., Schatz, M. C., and Salzberg S. L. (2010). Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, **11**(11):R116.

Kurtz, S., Narechania, A., Stein, J., and Ware, D. (2008). A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, **9**(1), 517.

Kurtz, S., Phillippy, A., Delcher, A. L., Smoot, M., Shumway, M., Antonescu, C., and Salzberg, S. L. (2004) Versatile and open software for comparing large genomes. *Genome Biology*, **5**(2), R12.

Li, Y. and Yan, X. (2014). MSPKmerCounter: A fast and memory efficient approach for k-mer counting. Preprint at http://cs.ucsb.edu/∼yangli/papers/MSPKmerCounter.pdf.

Marçais, G. and Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, **27**(6), 764–770.

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

Miller, J. R., Delcher, A. L., Koren, S., Venter, E., Walenz, B., Brownley, A., Johnson, J., Li, K., Mobarry, C. M., and Sutton, G. G. (2008). Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, **24**(24), 2818–2824.

Putze, F., Sanders, P., and Singler, J. (2009). Cache-, hash-and space-efficient Bloom filters. *ACM Journal of Experimental Algorithms*, **14**.

Rizk, G., Lavenier, D., and Chikhi, R. (2013). DSK: k-mer counting with very low memory usage. *Bioinformatics*, **29**(5), 652–653.

Roberts, M., Hayes, W., Hunt, B. R., Mount, S. M., and Yorke, J. A. (2004). Reducing storage requirements for biological sequence comparison. *Bioinformatics*, **20**(18), 3363–3369.

Roberts, M., Hunt, B. R., Yorke, J. A., Bolanos, R. A., and Delcher, A. L. (2004). A preprocessor for shotgun assembly of large genomes. *Journal of Computational Biology*, **11**(4), 734–752.

Roy, R. S., Bhattacharya, D., and Schliep, A. (2014). Turtle: Identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, doi:10.1093/bioinformatics/btu132.

Satish, N., Kim, C., Chhugani, J., Nguyen, A.D., Lee, V.W., Kim, D., and Dubey, P. (2010). Fast Sort on CPUs and GPUs. A Case for Bandwidth Oblivious SIMD Sort. *Proc. of the 2010 Int. Conf. on Management of data*, pp. 351–362.

Schuepbach, T., Pagni, P., Bridge, A., Bougueleret, L., Xenarios, I., and Cerutti, L. (2013). pfsearchV3: a code acceleration and heuristic to search PROSITE profiles. *Bioinformatics*, **29**(9), 1215–1217.

Sebastião, N., Encarnação, G., and Roma, N. (2012). Implementation and performance analysis of efficient index structures for DNA search algorithms in parallel platforms. *Concurrency and Computation: Practice and Experience*, doi: 10.1002/cpe.2970.

Wood, D. E., Salzberg, S. L. (2014). Kraken: Ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, **15**(3), R46.

Supplementary material for the paper

KMC 2: Fast and resource-frugal k-mer counting

by

Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz

1 KMC USAGE

KMC 2 program constructs a database of statistics for input set of FASTQ files. This database can then be used from other software: directly via KMC API (described in Section 2) or by reading a textual file containing a list of k-mers and their related counters. This textual file can be obtained for a database by KMC-dump program, that is presented in Section 3 as a sample application of our KMC API. Section 4 describes the database format in detail for those interested in the low-level access to the data. Section 5 contains additional experimental results and a description of the parameters of execution of the examined programs. Section 6 contains description of how the automatic setting of parameters of KMC works.

As this document is in part a technical documentation of KMC, some parts of it (e.g., API, command-line parameters) are highly similar to the supplement of our previous paper: S. Deorowicz, A. Debudaj-Grabysz, Sz. Grabowski: Disk-based k-mer counting on a PC, *BMC Bioinformatics*, 14, 160 (2013). The mentioned paper described the previous version of the current tool, i.e., KMC 1.

Below we describe in detail the parameters and options of the KMC command-line tool, in version 2.0.

The general syntax is:
kmc [options] <input_file_name> <output_file_name> <working_directory>
or:
kmc [options] <@input_file_names> <output_file_name> <working_directory>

where the parameters are:

- input_file_name — a single file in FASTQ format (gzipped or not),
- @input_file_names — a file name with list of input files in FASTQ format (gzipped or not),
- output_file_name — the output database file; if such a file exists, it will be overwritten.

The configuration options comprise:

- -v — verbose mode (shows all parameter settings); default: false,
- -k<len> — k-mer length, k from 1 to MAX_K; default: 25,
- -m<size> — max amount of RAM in GB (from 4 to 1024); default: 12,
- -p<par> — set signature length (from 5 to 7); default: 7,
- -f[a/q/m] — input in FASTA format (-fa), FASTQ format (-fq) or multi FASTA (-fm); default: FASTQ,
- -q[value] — use Quake's compatible counting with [value] representing lowest quality; default: 33,
- -ci<value> — exclude k-mers occurring less than <value> times; default: 2,
- -cs<value> — maximal value of a counter; default: 255,
- -cx<value> — exclude k-mers occurring more of than <value> times; default: 1e9,
- -b — turn off transformation of k-mers into canonical form,
- -r — turn on RAM-only mode,
- -sf<value> — number of FASTQ reading threads,
- -sp<value> — number of splitting threads,
- -so<value> — number of sorter threads,
- -sr<value> — number of threads per single sorter,
- -t<value> — total number of threads.

The parameters -sf<value>, -sp<value>, -so<value>, and -sr<value> concern the internal work of KMC, i.e., their settings may affect the program's processing speed, but won't change its output. Not setting at least one parameter from this group makes KMC ignore them all. The parameter -t<value> sets total number of threads (including fastq readers, splitters, sorters and threads per single sorter, but NOT including disk writer, bin reader and main KMC thread). If -t<value>, -sf<value>, -sp<value>, -so<value>, and -sr<value> are specified the -t<value> is ignored. Setting the parameter -r causes all computations are performed using RAM memory, without using disk space (memory usage may exceed limit).

Here are some usage examples.

kmc -k27 -m24 NA19238.fastq NA.res \data\kmc_tmp_dir\
kmc -k27 -q -m24 @files.lst NA.res \data\kmc_tmp_dir\

2 API

In this section we describe two classes, CKmerAPI and CKMCFile. They can be used to obtain access to the databases produced by KMC program.

2.1 CKmerAPI class

This class represents a k-mer. Its key methods are:

- CKmerAPI(uint32 length = 0) — constructor, that creates the array kmer_data of appropriate size,
- CKmerAPI(const CKmerAPI &kmer) — copy constructor,
- char get_symbol(unsigned int pos) — returns k-mer's symbol at a given position (0-based),
- std::string to_string() — converts k-mer to string, using the alphabet ACGT,
- void to_string(char *str) — converts k-mer to string, using the alphabet ACGT; the function assumes that enough memory was allocated,
- void to_string(str::string &str) — converts k-mer to string, using the alphabet ACGT,
- bool from_string(std::string &str) — converts string (from alphabet ACGT) to k-mer,
- CKmerAPI() — destructor, releases the content of kmer_data array,
- overloaded operators: =, ==, $<$.

2.2 CKMCFile class

This class handles a k-mer database. Its key methods are:

- CKMCFile() — constructor,
- bool OpenForRA(std::string file_name) — opens two files: file_name with added extension ".kmc_pre" and ".kmc_suf", reads their whole content to enable random access (in memory), and then closes them,
- bool OpenForListing(std::string file_name) — opens the file file_name with added extension ".kmc_pre" and allows to read the k-mers one by one (whole database is not loaded into memory),
- bool ReadNextKmer(CKmerAPI &kmer, float &count) — reads next k-mer to kmer and updates its count; the return value is bool; true as long as not eof-of-file (available only when database is opened in listing mode),
- bool Close() — if the file was opened for random access, the allocated memory for its content is released; if the file was opened for listing, the allocated memory for its content is released and the ".kmer" file is closed,
- bool SetMinCount(uint32 x) — set the minimum counter value for k-mers; if a k-mer has count below x, it is treated as non-existent,
- uint32 GetMinCount(void) — returns the value (uint32) set with SetMinCount,
- bool SetMaxCount(uint32 x) — set the maximum counter value for k-mers; if a k-mer has count above x, it is treated as non-existent,
- uint32 GetMaxCount(void) — returns the value (uint32) set with SetMaxCount,
- uint64 KmerCount(void) — returns the number of k-mers in the database (available only for databases opened in random access mode),
- uint32 KmerLength(void) — returns the k-mer length in the database (available only for databases opened in random access mode),
- bool RestartListing(void) — sets the cursor for listing k-mers from the beginning of the file (available only for databases opened in listing mode). The method OpenForListing(std::string file_name) invokes it automatically, but it can be also called by a user,
- bool Eof(void) — returns true if all k-mers have been listed,
- bool CheckKmer(CKmerAPI &kmer, float &count) — returns true if kmer exists in the database and set its count if the answer is positive (available only for databases opened in random access mode),
- bool IsKmer(CKmerAPI &kmer) — returns true if kmer exists (available only for databases opened in random access mode),
- void ResetMinMaxCounts(void) — sets min_count and max_count to the values read from the database,
- bool Info(uint32 &_kmer_length, uint32 &_mode, uint32 &_counter_size, uint32 &_lut_prefix_length, uint32 &_signature_len, uint32 &_min_count, uint32 &_max_count, uint64 &_total_kmers) — gets current parameters from the k-mer database,
- CKMCFile() — destructor.

3 EXAMPLE OF API USAGE

The kmc_dump application (Figs. 8 and 9) shows how to list and print k-mers with at least min_count and at most max_count occurrences in the database. Fig. 8 presents parsing the command-line parameters, including -ci<value> and -cx<value>. Input and output file names are also expected. The code in Fig. 9 is for actual database handling. This database is represented by a CKMCFile object, which opens an input file for k-mer listing (the method bool OpenForListing(std::string file_name) is invoked). The parameter of the method SetMinCount (SetMaxCount) must be not smaller (not greater) than the corresponding parameter -ci (-cx) with which KMC was invoked (otherwise, nothing will be listed). The listed k-mers are in the form like:
AAACACCGT\t<value>
where the first part is the k-mer in natural representation, which is followed by a tab character, and its associated value (integer or float). (Such format is compatible with Quake, a widely used tool for sequencing error correction.) Note that, if needed, one can easily modify the output format, changing the lines 39 and 41 in Fig. 9.

For performance reasons, the KMC package contains two variants of the dump program. The first one, presented below, is the kmc_dump_sample program. The second variant, kmc_dump, is essentially the same, the only difference is the way the counters are printed. Instead of the fprintf function we used much faster way of converting numbers into the textual form. Thus, in real applications the kmc_dump variant should be used.

```
#include <iostream>
#include "../kmc_api/kmc_file.h"

void print_info(void);

int _tmain(int argc, char* argv[])
{
  CKMCFile kmer_database;
  int i;
  uint32 min_count_to_set = 0;
  uint32 max_count_to_set = 0;
  std::string input_file_name;
  std::string output_file_name;

  FILE * out_file;
  //-------------------------------------------------------------
  // Parse input parameters
  //-------------------------------------------------------------
  if(argc < 3)
  {
    print_info();
    return EXIT_FAILURE;
  }

  for(i = 1; i < argc; ++i)
  {
    if(argv[i][0] == '-')
    {
      if(strncmp(argv[i], "-ci", 3) == 0)
        min_count_to_set = atoi(&argv[i][3]);
      else if(strncmp(argv[i], "-cx", 3) == 0)
          max_count_to_set = atoi(&argv[i][3]);
    }
    else
      break;
  }

  if(argc - i < 2)
  {
    print_info();
    return EXIT_FAILURE;
  }

  input_file_name = std::string(argv[i++]);
  output_file_name = std::string(argv[i]);

  if((out_file = fopen (output_file_name.c_str(),"wb")) == NULL)
  {
    print_info();
    return EXIT_FAILURE;
  }

  setvbuf(out_file, NULL ,_IOFBF, 1 << 24);

  ...
```

Fig. 8. First part of kmc_dump_sample application

```
    //------------------------------------------------------------------
    // Open kmer database for listing and print kmers within min_count and max_count
    //------------------------------------------------------------------

    if (!kmer_database.OpenForListing(input_file_name))
    {
      print_info();
      return EXIT_FAILURE ;
    }
    else
    {
      uint32 _kmer_length;
      uint32 _mode;
      uint32 _counter_size;
      uint32 _lut_prefix_length;
      uint32 _signature_len;
      uint32 _min_count;
      uint32 _max_count;
      uint64 _total_kmers;

      kmer_database.Info(_kmer_length, _mode, _counter_size, _lut_prefix_length, _signature_len,
        _min_count, _max_count, _total_kmers);

      float counter;
      std::string str;

      CKmerAPI kmer_object(_kmer_length);

      if(min_count_to_set)
        if (!(kmer_database.SetMinCount(min_count_to_set)))
          return EXIT_FAILURE;
      if(max_count_to_set)
        if (!(kmer_database.SetMaxCount(max_count_to_set)))
          return EXIT_FAILURE;

      while(kmer_database.ReadNextKmer(kmer_object, counter))
      {
        kmer_object.to_string(str);

        if(_mode)
          fprintf(out_file, "%s\t%f\n", str.c_str(), counter);
        else
          fprintf(out_file, "%s\t%d\n", str.c_str(), (int)counter);
      }

      fclose(out_file);
    }

    return EXIT_SUCCESS;
}

// ------------------------------------------------------------------
// Print execution options
// ------------------------------------------------------------------
void print_info(void)
{
    std::cout << "KMC dump ver. " << KMC_VER << " (" << KMC_DATE << ")\n";
    std::cout << "\nUsage:\nkmc_dump [options] <kmc_database> <output_file>\n";
    std::cout << "Parameters:\n";
    std::cout << "<kmc_database> - kmer_counter's output\n";
    std::cout << "Options:\n";
    std::cout << "-ci<value> - print k-mers occurring less than <value> times\n";
    std::cout << "-cx<value> - print k-mers occurring more of than <value> times\n";
};
```

Fig. 9. Second part of kmc_dump_sample application

4 DATABASE FORMAT

The KMC application creates output files with two extensions:

- .kmc_pre — with information on k-mer prefixes (plus some other data),
- .kmc_suf — with information on k-mer suffixes and the related counters.

All integers in the KMC output files are stored in LSB (least significant byte first) byte order.

4.1 The .kmc_pre file structure

The .kmc_pre file contains, in order, the following data:

- [marker],
- [prefixes],
- [map],
- [header],
- [header position],
- [marker] (another copy, to signal the file is not truncated).

[marker] 4 bytes with the letters: KMCP.

[header position] The integer consisting of the last 4 bytes in the file (before end KMCP marker). It contains the relative position of the beginning of the field [header]. After opening the file, one should do the following:

1. Read the first 4 bytes and check if they contain the letters KMCP.
2. Read the last 4 bytes and check if they contain the letters KMCP.
3. Jump to position 8 bytes back from end of file and read the header position x.
4. Jump to position $x + 8$ bytes back from end of file and read the header.
5. Read [data].

[header] The header contains fields describing the file .kmc_pre:

- uint32 kmer_length — k-mer length,
- uint32 mode — mode: 0 (occurrence counters) or 1 (quality-aware counters),
- uint32 counter_size — counter field size: for mode 0 it is 1, 2, 3, or 4; for mode 1 it is always 4,
- uint32 lut_prefix_length — the length (in symbols) of the prefix cut off from k-mers; it is invariant of the scheme that 4 divides $(kmer\_length - lut\_prefix\_length)$,
- uint32 signature_length — the length (in symbols) of the signature,
- uint32 min_count — minimum number of k-mer occurrences to write in the database (if the counter is smaller, the k-mer data are not written),
- uint32 max_count — maximum number of k-mer occurrences to write in the database (if the counter is greater, the k-mer data are not written),
- uint64 total_kmers — total number of k-mers in the database,
- uint32 tmp[7] — not used in the current version,
- uint32 KMC_VER — version of KMC software (for KMC 2 this value is equal to 0x200).

[map]

There is an array of uint32 elements, of size $4^{signature\_length} + 1$. This array is used to identify position of proper prefixes' array stored in [prefixes] region. For example, if the queried k-mer is ATACGACAAATG and $signature\_length = 5$, its signature is ACGAC (as it is the smallest 5-mer which satisfies conditions of being a signature). DNA symbols are encoded as follows: A $\rightarrow$ 0, C $\rightarrow$ 1, G $\rightarrow$ 2, T $\rightarrow$ 3, so ACGAC is equal to 97 (since $0 \cdot 2^8 + 1 \cdot 2^6 + 2 \cdot 2^4 + 0 \cdot 2^2 + 1 \cdot 2^0 = 97$). In this case we look into "map" at position 97 to get the id of related prefixes' array.

[prefixes]

This region contains a number of prefixes' arrays (typically hundreds of them) of uint64 elements. Each array is of size $4^{lut\_prefix\_length}$. The last prefixes' array is followed by an additional uint64 element being a guard to make the reading process simpler. The total number of prefixes' arrays can be easily calculated (as start and end position are given, size of one array is also known). The element at position x in prefixes' array for given signature s points to a record in .kmc_suf file. This record contains the first suffix of k-mer with prefix x and signature s (the position of the last record can be obtained by decreasing the value at $x + 1$ in prefixes' array by 1).

Using the example from the previous section, the start position of prefixes' array for k-mer ATACGACAAATG should be calculated as: $4 + 97 \cdot 4^{lut\_prefix\_length} \cdot 8$ (marker + equivalent of ACGAC signature · no. of elements in each array · size of element in prefix array). The next step is to cut off the prefix of length equal to lut_prefix_length from the queried k-mer. Let us assume $lut\_prefix\_length = 4$, and then the prefix is ATAC whose equivalent is 49. The element at position 49 in the related prefixes' array (pointed by signature 97) is the position of the first record in .kmc_suf file which contains a k-mer with prefix ATAC and with signature ACGAC. Let us suppose this position is 1523, then we look at position 50 in prefixes' array (say, it contains 1685). This means that .kmc_suf file stores the suffixes of k-mers with prefix ATAC and signature ACGAC in the records from 1523 to $1685 - 1$. Having got this range, we can now apply binary search for the suffix GACAAATG.

4.2 The.kmc_suf file structure

The .kmc_suf file contains, in order, the following data:

- [marker],
- [data],
- [marker] (another copy, to signal the file is not truncated).

The k-mers are stored with their leftmost symbol first, packed into bytes. For example, CCACAAAT is represented as 0x51 (for CCAC), 0x03 (for AAAT). Integers are stored according to the LSB (little endian) byte order, floats are stored in the same way as they are stored in the memory.

[marker] 4 bytes with the letters: KMCS.

[data] An array record_t records[total_kmers].

total_kmers is taken from the .kmc_pre file.

record_t is a type representing a k-mer. Its first field is the k-mer suffix string, stored on $(kmer\_length - lut\_prefix\_length)/4$ bytes. The next field is $counter\_size$, with the number of bytes used by the counter, which is either a $1 \ldots 4$-byte integer, or a 4-byte float.

5 EXPERIMENTAL RESULTS

5.1 Test platforms

K-mer Counter (KMC), was implemented in C++11, using gcc compiler (version 4.8.3) for the linux build and Microsoft Visual Studio 2013 for the Windows build.

The configuration of the test machine was:

- CPU: Intel i7 4930 (6-cores clocked at 3.4 GHz),
- RAM: 64 GB RAM (clocked at 1833 MHz),
- HDD: 2 drives Seagate Constellation ES.3 3 TB each in RAID 0; buffered transfers reported by hdparm -t: 355 MB/s.
- SSD: Samsung 840 Evo 1 TB; buffered transfers reported by hdparm -t: 510 MB/s.

5.2 Datasets

5.2.1 F. vesca The files were downloaded from the following URLs:
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030576/SRR072006.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030576/SRR072007.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030577/SRR072008.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030577/SRR072009.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030578/SRR072013.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030578/SRR072014.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030578/SRR072029.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030575/SRR072005.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030575/SRR072010.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030575/SRR072011.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030575/SRR072012.fastq.bz2
Then they were decompressed to a single fv.fastq file.

5.2.2 G. gallus The files were downloaded from the following URLs:
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030308/SRX043656/SRR105788_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030308/SRX043656/SRR105788_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030309/SRX043656/SRR105789_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030309/SRX043656/SRR105789_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030312/SRX043656/SRR105792_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030312/SRX043656/SRR105792_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030314/SRX043656/SRR105794.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030314/SRX043656/SRR105794_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030314/SRX043656/SRR105794_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036382/SRX043656/SRR197985.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036382/SRX043656/SRR197985_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036382/SRX043656/SRR197985_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036383/SRX043656/SRR197986.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036383/SRX043656/SRR197986_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036383/SRX043656/SRR197986_2.fastq.bz2
Then they were decompressed to a single gg.fastq file. The files were also re-compressed to gzip format for the experiments with k-mer counting of gzipped files.

5.2.3 M. balbisiana The files were downloaded from the following URLs:
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA098/SRA098922/SRX339427/SRR956987.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA098/SRA098922/SRX339427/SRR957627.fastq.bz2
Then they were decompressed to a single mb.fastq file.

5.2.4 H. sapiens *1* The files were downloaded from the following URL:
ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data/HG02057/sequence_read/
Then they were decompressed to a single hs1.fastq file.

5.2.5 H. sapiens *2* The FASTQ files (48 files) were downloaded from the following URL:
http://www.ebi.ac.uk/ena/data/view/ERA015743
Then they were decompressed to a single hs2.fastq file. The file hs2_files contains list of gzipped files of this individual.

5.3 Parameters of programs

Jellyfish Jellyfish (ver. 2.1.3) requires to give as a parameter the expected number of counted k-mers. In all experiments we set this value to be about 10% larger than the number of k-mers reported by KMC.

Command lines:

```
./jellyfish count -m 28 -C -s 300M -t 12 -L 2 -o jelly2 fv.fastq
./jellyfish count -m 55 -C -s 400M -t 12 -L 2 -o jelly2 fv.fastq
./jellyfish count -m 28 -C -s 1200M -t 12 -L 2 -o jelly2 gg.fastq
./jellyfish count -m 55 -C -s 1200M -t 12 -L 2 -o jelly2 gg.fastq
./jellyfish count -m 28 -C -s 1G -t 12 -L 2 -o jelly2 mb.fastq
./jellyfish count -m 55 -C -s 1200M -t 12 -L 2 -o jelly2 mb.fastq
./jellyfish count -m 28 -C -s 3G -t 12 -L 2 -o jelly2 hs1.fastq
./jellyfish count -m 55 -C -s 3G -t 12 -L 2 -o jelly2 hs1.fastq
./jellyfish count -m 28 -C -s 3G -t 12 -L 2 -o jelly2 hs2.fastq
./jellyfish count -m 55 -C -s 3G -t 12 -L 2 -o jelly2 hs2.fastq
```

KAnalyze KAnalyze (ver. 0.9.5) does not allow to count k-mers for $k > 32$, so only a single value of k was used in the tests. Since KAnalyze documentation does not say how to divide the threads among "k-mer step" and "split step" we allocated 6 threads for both steps (`-l` and `-d` parameters).

Command lines:

```
java -jar ./kanalyze.jar count -d 6 -f fastq -k 28 -l 6 fv.fastq
java -jar ./kanalyze.jar count -d 6 -f fastq -k 28 -l 6 gg.fastq
java -jar ./kanalyze.jar count -d 6 -f fastq -k 28 -l 6 mb.fastq
java -jar ./kanalyze.jar count -d 6 -f fastq -k 28 -l 6 hs1.fastq
java -jar ./kanalyze.jar count -d 6 -f fastq -k 28 -l 6 hs2.fastq
```

DSK DSK (ver. 1.6066) was executed with default parameters that means 6 GB limit of RAM. The k-mers occurring less than 2 times were excluded.

Command lines:

```
./dsk32 fv.fastq 28 -t 2 -m 6144 -o o_dsk
./dsk64 fv.fastq 55 -t 2 -m 6144 -o o_dsk
./dsk32 gg.fastq 28 -t 2 -m 6144 -o o_dsk
./dsk64 gg.fastq 55 -t 2 -m 6144 -o o_dsk
./dsk32 mb.fastq 28 -t 2 -m 6144 -o o_dsk
./dsk64 mb.fastq 55 -t 2 -m 6144 -o o_dsk
./dsk32 hs1.fastq 28 -t 2 -m 6144 -o o_dsk
./dsk64 hs1.fastq 55 -t 2 -m 6144 -o o_dsk
./dsk32 hs2.fastq 28 -t 2 -m 6144 -o o_dsk
./dsk64 hs2.fastq 55 -t 2 -m 6144 -o o_dsk
```

Turtle The program scTurtle (ver. 0.3) was used to calculate the k-mers and their counts. The documentation says that the number of threads should be a prime, so for our 12-virtual cores system we used 11 threads. The expected number of k-mers was set to be about 10% larger than the exact value (calculated by KMC).

Command lines:

```
./scTurtle32 -f fv.fastq -o turtle -k 28 -t 11 -n 400000000
./scTurtle64 -f fv.fastq -o turtle -k 55 -t 11 -n 400000000
./scTurtle32 -f gg.fastq -o turtle -k 28 -t 11 -n 1150000000
./scTurtle64 -f gg.fastq -o turtle -k 55 -t 11 -n 1150000000
./scTurtle32 -f mb.fastq -o turtle -k 28 -t 11 -n 1100000000
./scTurtle64 -f mb.fastq -o turtle -k 28 -t 11 -n 1100000000
./scTurtle32 -f hs1.fastq -o turtle -k 28 -t 11 -n 3000000000
./scTurtle64 -f hs1.fastq -o turtle -k 28 -t 11 -n 3000000000
./scTurtle32 -f hs2.fastq -o turtle -k 28 -t 11 -n 3000000000
./scTurtle64 -f hs2.fastq -o turtle -k 28 -t 11 -n 3000000000
```

MSPKmerCounter MSPKmerCounter (ver. 0.10.0) was used with minimizer length (10) and number of bins (1000) suggested in the original paper.

Command lines:

```
java -jar ./Partition.jar -in fv.fastq -k 28 -L 353 -NB 1000 -p 10 -t 12
java -jar ./Count32.jar -t 12 -k 28 -NB 1000

java -jar ./Partition.jar -in fv.fastq -k 55 -L 353 -NB 1000 -p 10 -t 12
java -jar ./Count64.jar -t 12 -k 55 -NB 1000

java -jar ./Partition.jar -in gg.fastq -k 28 -L 100 -NB 1000 -p 10 -t 12
java -jar ./Count32.jar -t 12 -k 28 -NB 1000

java -jar ./Partition.jar -in gg.fastq -k 55 -L 100 -NB 1000 -p 10 -t 12
java -jar ./Count64.jar -t 12 -k 55 -NB 1000

java -jar ./Partition.jar -in mb.fastq -k 28 -L 101 -NB 1000 -p 10 -t 12
java -jar ./Count32.jar -t 12 -k 28 -NB 1000

java -jar ./Partition.jar -in mb.fastq -k 55 -L 101 -NB 1000 -p 10 -t 12
java -jar ./Count64.jar -t 12 -k 55 -NB 1000

java -jar ./Partition.jar -in hs1.fastq -k 28 -L 100 -NB 1000 -p 10 -t 12
java -jar ./Count32.jar -t 12 -k 28 -NB 1000

java -jar ./Partition.jar -in hs1.fastq -k 55 -L 100 -NB 1000 -p 10 -t 12
java -jar ./Count64.jar -t 12 -k 55 -NB 1000

java -jar ./Partition.jar -in hs2.fastq -k 28 -L 101 -NB 1000 -p 10 -t 12
java -jar ./Count32.jar -t 12 -k 28 -NB 1000

java -jar ./Partition.jar -in hs2.fastq -k 55 -L 101 -NB 1000 -p 10 -t 12
java -jar ./Count64.jar -t 12 -k 55 -NB 1000
```

KMC 1 KMC was executed with setting that k-mers occurring less than 2 times should not be counted. The parameter -p (prefix length) was set to 5 for all data sets except the smallest one.

Command lines:

```
./kmc1 -v -m16 -k28 -p4 fv.fastq res temp
./kmc1 -v -m16 -k55 -p4 fv.fastq res temp
./kmc1 -v -m16 -k28 -p5 gg.fastq res temp
./kmc1 -v -m16 -k55 -p5 gg.fastq res temp
./kmc1 -v -m16 -k28 -p5 mb.fastq res temp
./kmc1 -v -m16 -k55 -p5 mb.fastq res temp
./kmc1 -v -m16 -k28 -p5 hs1.fastq res temp
./kmc1 -v -m16 -k55 -p5 hs1.fastq res temp
./kmc1 -v -m16 -k28 -p5 hs2.fastq res temp
./kmc1 -v -m16 -k55 -p5 hs2.fastq res temp
```

KMC 2 KMC was executed with setting that k-mers occurring less than 2 times should not be counted. The parameter -p (minimizer length) was set to 7 for all data sets.

Command lines (main tests):

```
./kmc2 -v -m12 -k28 -p7 fv.fastq res temp
./kmc2 -v -m12 -k28 -p7 fv.fastq res temp
./kmc2 -v -m6 -k28 -p7 fv.fastq res temp
./kmc2 -v -m6 -k28 -p7 fv.fastq res temp
./kmc2 -v -m12 -k28 -p7 gg.fastq res temp
./kmc2 -v -m12 -k28 -p7 gg.fastq res temp
./kmc2 -v -m6 -k28 -p7 gg.fastq res temp
./kmc2 -v -m6 -k28 -p7 gg.fastq res temp
./kmc2 -v -m12 -k28 -p7 mb.fastq res temp
./kmc2 -v -m12 -k28 -p7 mb.fastq res temp
./kmc2 -v -m6 -k28 -p7 mb.fastq res temp
```

```
./kmc2 -v -m6 -k28 -p7 mb.fastq res temp
./kmc2 -v -m12 -k28 -p7 hs1.fastq res temp
./kmc2 -v -m12 -k28 -p7 hs1.fastq res temp
./kmc2 -v -m6 -k28 -p7 hs1.fastq res temp
./kmc2 -v -m6 -k28 -p7 hs1.fastq res temp
./kmc2 -v -m12 -k28 -p7 hs2.fastq res temp
./kmc2 -v -m12 -k28 -p7 hs2.fastq res temp
./kmc2 -v -m6 -k28 -p7 hs2.fastq res temp
./kmc2 -v -m6 -k28 -p7 hs2.fastq res temp
```

Command lines (gzipped files):

```
./kmc2 -v -m12 -k28 -p7 @hs2_files res temp
./kmc2 -v -m12 -k55 -p7 @hs2_files res temp
```

where `hs2_files` contains list of gzipped FASTQ files of `hs2` data set.

Command lines (thread tests):

```
taskset -c 0 ./kmc2 -v -m12 -k28 -p7 -t2 @gg_files res temp
taskset -c 0 ./kmc2 -v -m12 -k28 -p7 -t2 @gg_files res temp
./kmc2 -v -m12 -k55 -p7 -t2 @gg_files res temp
./kmc2 -v -m12 -k28 -p7 -t2 @gg_files res temp
./kmc2 -v -m12 -k55 -p7 -t2 @gg_files res temp
./kmc2 -v -m12 -k28 -p7 -t3 @gg_files res temp
./kmc2 -v -m12 -k55 -p7 -t3 @gg_files res temp
./kmc2 -v -m12 -k28 -p7 -t4 @gg_files res temp
./kmc2 -v -m12 -k55 -p7 -t4 @gg_files res temp
./kmc2 -v -m12 -k28 -p7 -t5 @gg_files res temp
./kmc2 -v -m12 -k55 -p7 -t5 @gg_files res temp
./kmc2 -v -m12 -k28 -p7 -t6 @gg_files res temp
./kmc2 -v -m12 -k55 -p7 -t6 @gg_files res temp
./kmc2 -v -m12 -k28 -p7 -t8 @gg_files res temp
./kmc2 -v -m12 -k55 -p7 -t8 @gg_files res temp
./kmc2 -v -m12 -k28 -p7 -t10 @gg_files res temp
./kmc2 -v -m12 -k55 -p7 -t10 @gg_files res temp
./kmc2 -v -m12 -k28 -p7 -t12 @gg_files res temp
./kmc2 -v -m12 -k55 -p7 -t12 @gg_files res temp
```

where `gg_files` contains list of gzipped FASTQ files of `gg` data set.

Since KMC 2 does not allow to specify less than 2 threads to measure the speed of KMC for a single thread scenario we allowed to use a single core by using the linux taskset command.

5.4 Results

The results for additional data sets (the ones that are not included in the main part of the paper) are given below.

Table 8. k-mers counting results for *F. vesca*. MSPKC fails, probably due to the variable length of reads in the dataset.

Algorithm	$k = 28$ RAM	Disk	Time	$k = 55$ RAM	Disk	Time
			SSD			
Jellyfish 2	9	0	133	39	0	243
KAnalyze	9	33	345		*unsupported k*	
DSK	6	12	141	6	13	298
Turtle	17	0	133	26	0	175
MSPKC		*failed*			*failed*	
KMC 1	13	17	84	17	41	243
KMC 2 (12GB)	7	4	45	12	3	59
KMC 2 (6GB)	6	4	33	6	3	60
			HDD			
Jellyfish 2	9	0	133	39	0	245
DSK	6	12	147	6	13	308
Turtle	17	0	135	26	0	178
KMC 1	11	17	120	17	41	245
KMC 2	7	4	58	12	3	61

Table 9. k-mers counting results for *H. sapiens* 1.

Algorithm	$k = 28$ RAM	Disk	Time	$k = 55$ RAM	Disk	Time
			SSD			
Jellyfish 2	62	0	2,013		*out of memory*	
KAnalyze		*out of disk ($>$ 650 GB)*			*unsupported k*	
DSK	6	192	3,485	6	236	4,475
Turtle		*out of memory*			*out of memory*	
MSPKC	17	286	10,032		*out of time ($>$ 10 hours)*	
KMC 1	17	251	1,930	17	426	3,788
KMC 2 (12GB)	12	64	1,010	12	44	1,251
KMC 2 (6GB)	6	64	1,013	8	44	1,397
			HDD			
Jellyfish 2	62	0	2,209		*out of memory*	
DSK	6	192	10,667	6	236	13,550
MSPKC	17	286	13,444		*out of time ($>$ 10 hours)*	
KMC 1	17	251	3,296	17	426	5,136
KMC 2	12	64	1,417	12	44	1,651

6 AUTOMATIC SETTING OF PARAMETERS IN KMC

The automatic setting of parameters mechanism tries to allocate the available resources (i.e., CPU cores) in the best possible way. The optimal number of threads for the parts of the algorithm is, however, hard to obtain, since it depends on many things, like the compression method of input files, the speed of disks, etc. Thus, our automatic mechanism is obviously suboptimal, nevertheless, experiments show that it performs reasonably well. If the results are unsatisfactory, the KMC 2 user can specify these parameters from command line.

The most important factor of the mechanism is the number of available cores (possibly overridden if the user specifies it with `-t` parameter). FASTQ readers, splitters, sorters and sorting threads per single sorter are the most important threads of KMC 2. To set an exact number of those threads, **all** `-s?` parameters must be specified (if one is omitted, the rest of them is ignored).

The automatic setting of parameters for the first stage works as follow. If the input files are in plain text format (not compressed), there is one FASTQ reader thread. Otherwise the number of FASTQ reader threads is equal to the number of large input files ("large" means here the ones whose size is greater than 5% of the size of the largest file), but not more than half of the number of cores. After that the number of "free" cores (not assigned yet) is set as the number of splitting threads. In the second stage the memory requirements for each bin are known and the following steps are performed. Bins are sorted by their size in a non-increasing order. The number of sorter threads is calculated as $\lfloor M_2/B_{10} \rfloor$, where M_2 is total amount of memory available for the second stage, B_{10} is the size of the bin for which `10%` of bins are bigger. The number of sorting threads per single sorter is equal to the number of cores divided by the number of sorter threads (as the result may not be integer, some sorters have one sorting thread more than others, e.g., if there are 7 sorters and 10 threads to allocate, 3 sorters would run 2 sorting threads and 4 sorters only 1 sorting thread).