Bioinformatics, 31(10), 2015, 1569–1576
doi: 10.1093/bioinformatics/btv022
Advance Access Publication Date: 20 January 2015
Original Paper

Sequence analysis

KMC 2: fast and resource-frugal *k*-mer counting

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

[1]Institute of Informatics, Silesian University of Technology, Akademicka 16, 44-100 Gliwice and [2]Institute of Applied Computer Science, Lodz University of Technology, Al. Politechniki 11, 90-924 Łódź, Poland

*To whom correspondence should be addressed.
Associate Editor: John Hancock

Received on July 8, 2014; revised on January 10, 2015; accepted on January 12, 2015

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 about twice faster than the strongest competitors (Jellyfish 2, KMC 1), using about 12 GB (or less) of RAM. 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 counts the 28-mers of a human reads collection with 44-fold coverage (106 GB of compressed size) in about 20 min, on a 6-core Intel i7 PC with an solid-state disk.
Availability and implementation: KMC 2 is freely available at http://sun.aei.polsl.pl/kmc.
Contact: sebastian.deorowicz@polsl.pl
Supplementary information: Supplementary data are available at *Bioinformatics* online.

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 an HT. Although BF is a probabilistic mechanism, BFCounter applies it in a smart way, which does not produce counting errors. 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 1.1 GB of RAM (Chikhi *et al.*, 2014), whereas KMC is faster but typically uses about 11–16 GB of RAM. Turtle (Roy *et al.*, 2014) bears some similarities to BFCounter. The standard BF is there replaced with its cache-friendly variant (Putze *et al.*, 2009) and the HT 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 article, 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) and disk space (about 5 times) when compared with MSPKmerCounter. The second main novelty is the use of (k, x)-mers $(x > 0)$ for reduction of the amount of data to sort. Simply, instead of sorting some amount of *k*-mers, we sort a much smaller portion of $(k+x)$-mers and then obtain the statistics for *k*-mers in the post-processing 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 (HT in Jellyfish, BF 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 HT; after processing a bin, the entries from the HT are dumped to disk and the HT 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 than that in the case of whole input data.

This elegant idea allows to significantly reduce the disk space compared with 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 article, 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 canonical minimizers as signatures, 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. Roberts *et al.* (2004a, b) suggest remapping the ACGT alphabet to integers 1, 0, 3 and 2 for odd-numbered bases of *k*-mers and reverse the ordering for even-numbered bases. As they note that the letters C and G often occur less frequently than A and T, the proposed reordering tends to start minimizers with the valuable (in the sense of the significance of a match) letters C and G and the minimum k-mer is CGCGCG … In Roberts *et al.* (2004a), they also consider only the minimizers with total counts in the read collection below some threshold, e.g. 75 occurrences. Wood and Salzberg (2014) note that using standard minimizers in their metagenomic sequence classification would result in over-representation of low-complexity minimizer strings, implying longer search times. To prevent it and thus obtain a more even distribution of minimizers, they use the XOR operation to toggle half of the bits of each m-mer's canonical representation prior to comparing the m-mers to each other using lexicographical ordering.

Coming back to our minimizers' variant, as the experiments show (cf. experimental section 3 of the paper), the mentioned 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 with 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 necessary to store all the k-mers in explicit form. Then, the k-mers are either sorted or inserted into a HT or BF. 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, for some $0 \le x' \le x$, such that all the k-mers in their span are in the canonical form.

The idea is that instead of breaking super k-mers into k-mers (for sorting purposes), we break them into as few (k, x)-mers as possible in such way that the canonical form of each k-mer present in a super k-mer belongs to exactly one (k, x)-mer. 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 2000 (512 by default), some signatures are merged (i.e. the corresponding sequences are sent to the same bin). To decide which signatures to merge, in a pre-processing stage, KMC 2 reads a small fraction of the input data, builds a histogram of found signatures and finally merges the least frequent signatures (more details are given in the Supplementary Material).

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 canonical 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 canonical k-mers from (k,1)-mers.

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

1 it can be a (k,1)-mer of length k,
2–5 it can be a suffix of a $(k + 1)$-mer preceded by A, C, G or T,
6 it can be a prefix of some $(k + 1)$-mer.

Therefore, we conceptually split the array of (k,1)-mers into five non-overlapping, sorted subarrays: one (R_0) containing $(k + 0)$-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).

Now to collect the statistics of k-mers, we scan these six subarrays in parallel considering suffixes in case of R_A, R_C, R_G, R_T and prefixes in case of R_1. So, we have six pointers somewhere in R_* We compare the pointed elements, find the lexicographically smallest canonical k-mer among them and store it in the resulting array of statistics of canonical k-mers P if it is different than the recently added k-mer to P. Otherwise, we just increase the counter related to this canonical 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 Figure 3. Several FASTQ readers send input data chunks into a queue, handled then

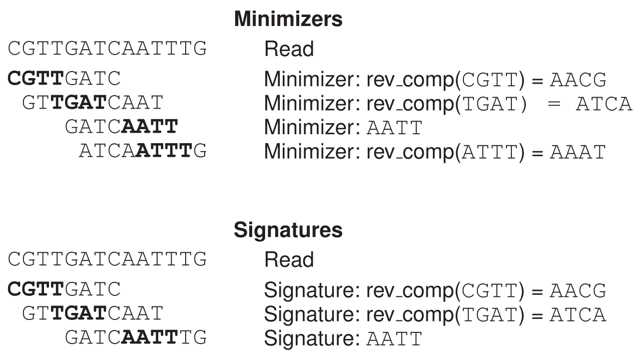
Minimizers
CGTTGATCAATTTG Read
CGTTGATC Minimizer: rev_comp(CGTT) = AACG
GTTGATCAAT Minimizer: rev_comp(TGAT) = ATCA
GATCAATT Minimizer: AATT
ATCAATTTG Minimizer: rev_comp(ATTT) = AAAT
Signatures
CGTTGATCAATTTG Read
CGTTGATC Signature: rev_comp(CGTT) = AACG
GTTGATCAAT Signature: rev_comp(TGAT) = ATCA
GATCAATTTG Signature: AATT

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

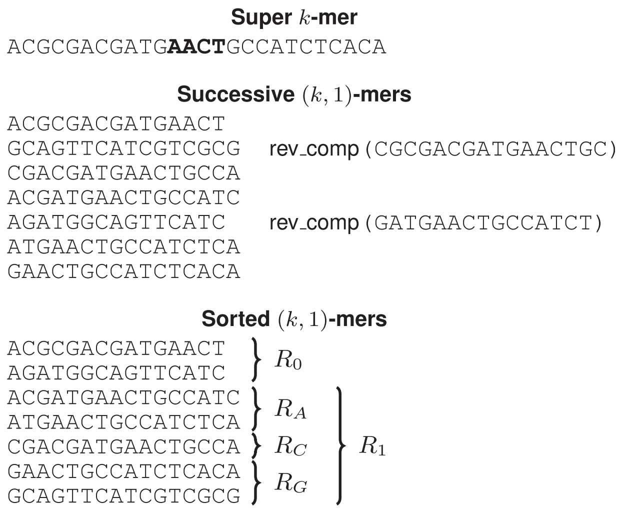

Fig. 2. Splitting a super k-mer into (k,1)-mers followed by sorting them. The assumed parameters are: $k = 15$, $m = 4$. The range R_T is empty (thus not shown). Note that the (k,1)-mers from the subarray R_0 are k-mers, the (k,1)-mer from the subarray R_A are $(k + 1)$-mers with the starting symbol A, etc.; the concatenation of R_A, R_C, R_G and R_T forms the (conceptual) subarray R_1

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, 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 in-memory mode in which the bins are simply stored in the main memory, which may be convenient for large datacenters.

The standard KMC 2 memory usage setting works only as a suggestion and not a strict limit. If a single bin needs more memory, KMC 2 will break the given limit (which can be observed in the experimental results). Nevertheless, KMC 2 can be run in the strict memory mode in which the given limit (no less, however, than 1 GB) cannot be exceeded. In this mode, larger bins are split into smaller ones, sorted one by one and dumped to disk. Finally, their sorted parts are collected from disk, merged and again written to disk. This increases the overall I/O, but the total disk usage is usually unchanged since the temporary files for sorting the large bins are created when most of bin files are already removed.

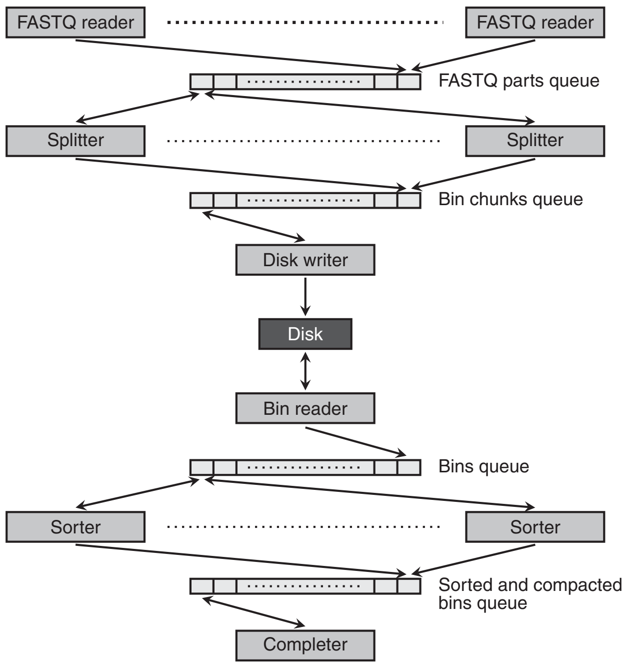

Fig. 3. A scheme of the parallel KMC algorithm

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 (Rizk *et al.*, 2013), Turtle (Roy *et al.*, 2014), MSPCounter (Li and Yan, 2014), KAnalyze (Audano and Vannberg, 2014) and KMC 1 (Deorowicz *et al.*, 2013). Each program was tested for two values of *k* (28 and 55) and in two hardware configurations: using conventional hard 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). Some of the experiments were also run on a single HDD (5 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.

In the experiments, we count only *k*-mers with counts at least 2, since the *k*-mers with a single occurrence in a read collection most likely contain erroneous base(s). As in some applications, all *k*-mers may be needed, we ran a preliminary KMC 2 test in such setting, with a SSD. We found out that the overhead in computation time is only up to 3% (mainly caused by increased I/O).

The comparison, presented in Tables 2–4 and Supplementary Tables S1 and S2, includes total computation time (in seconds), maximum RAM use and maximum disk use. RAM and disk use are given in GBs ($1\text{GB} = 10^9$ B). Time is wall-clock time in seconds. A test running longer than 10 h 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, *Homo sapiens* 2, occupies 312.9 GB on the same disk). Jellyfish 2 was tested twice, in the default and the BF-based mode with exact counts. Unfortunately, in the latter experiments, the amount of memory in our machine was often not enough, and this is why Jellyfish 2-BF results are shown only for two datasets.

Several conclusions can easily be drawn from the presented tables. Two of the competitors, KAnalyze and MSPKC, are clearly

Table 1. Characteristics of the datasets used in the experiments

Organism	Genome length	No. bases	FASTQ file size	No. files	Gzipped size	Average read length
F.vesca	210	4.5	10.3	11	3.5	353
G.gallus	1040	34.7	115.9	15	25.9	100
M.balbisiana	472	56.9	197.1	2	49.1	101
H.sapiens 1	3093	86.0	223.3	6	70.8	100
H.sapiens 2	3093	135.3	312.9	48	105.8	101

Number of bases are in Gbases. File sizes are in Gbytes (1 Gbyte = 10^9 bytes). Approximate genome lengths are in Mbases according to http://www.ncbi.nlm.nih.gov/genome/.

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

Algorithm	$k=28$			$k=55$		
	RAM	Disk	Time	RAM	Disk	Time
SSD						
Jellyfish 2	33	0	880	*Out of memory*		
Jellyfish 2-BF	35	19	2237	57	19	2180
KAnalyze	9	270	11 071	*Unsupported k*		
DSK	6	101	1325	6	94	1836
Turtle	48	0	1004	*Out of memory*		
MSPKC	17	114	3382	*Out of time (>10 h)*		
KMC 1	13	101	868	12	173	1792
KMC 2 (12 GB)	12	25	408	12	18	503
KMC 2 (6 GB)	6	25	431	6	18	562
KMC 2 (4 GB)	4	25	523	4	18	681
KMC 2 (in-mem)	33	0	343	27	0	466
HDD						
Jellyfish 2	33	0	915	*Out of memory*		
DSK	6	101	3600	6	94	4206
Turtle	48	0	1058	*Out of memory*		
MSPKC	17	114	4853	*Out of time (>10 h)*		
KMC 1	11	101	1320	12	173	2036
KMC 2 (12 GB)	12	25	587	12	18	656
KMC 2 (4 GB)	4	25	651	4	18	854

Timings in seconds and RAM/disk consumption in GB.

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

Algorithm	$k=28$			$k=55$		
	RAM	Disk	Time	RAM	Disk	Time
SSD						
Jellyfish 2	17	0	1080	26	0	853
Jellyfish 2-BF	*Out of memory*			56	30	2865
KAnalyze	9	354	8249	—	—	—
DSK	6	164	2356	6	138	2962
Turtle	46	0	1484	*Out of memory*		
MSPKC	10	185	8729	*Out of time (>10 h)*		
KMC 1	13	165	1229	15	279	2622
KMC 2 (12 GB)	12	41	755	12	29	834
KMC 2 (6 GB)	6	41	685	6	29	895
KMC 2 (4 GB)	4	41	833	4	29	896
KMC 2 (in-mem)	49	0	663	38	0	780
HDD						
Jellyfish 2	17	0	1115	26	0	881
DSK	6	164	6216	6	138	7228
Turtle	46	0	1498	*Out of memory*		
MSPKC	10	185	12 152	*Out of time (>10 h)*		
KMC 1	13	165	2194	15	279	3367
KMC 2 (12 GB)	12	41	960	12	29	1041
KMC 2 (4 GB)	4	41	1051	4	29	1207

Timings in seconds and RAM/disk consumption in GB.

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 h of processing. The only asset of KAnalyze and MSPKC we have found is their moderate memory use.

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

Algorithm	$k=28$			$k=55$		
	RAM	Disk	Time	RAM	Disk	Time
SSD						
Jellyfish 2	62	0	3212	*Out of memory*		
KAnalyze	*Out of disk (>650 GB)*			*Unsupported k*		
DSK	6	263	5487	6	256	7732
Turtle	*Out of memory*			*Out of memory*		
MSPKC	*Out of time (>10 h)*			*Out of time (>10 h)*		
KMC 1	17	396	2998	*Out of disk (>650 GB)*		
KMC 2 (12 GB)	12	101	1615	13	70	2038
KMC 2 (6 GB)	6	101	1706	13	70	2446
KMC 2 (4 GB)	4	101	1843	4	70	2802
HDD						
Jellyfish 2	62	0	3231	*Out of memory*		
DSK	6	263	18 493	6	256	22 432
KMC 1	17	396	4898	*Out of disk (>650 GB)*		
KMC 2 (12 GB)	12	101	2259	13	70	2640
KMC 2 (4 GB)	4	101	2707	4	70	3471

Timings in seconds and RAM/disk consumption in GB.

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 in its default mode 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.

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 three times for each k: with standard memory use (12 GB) and with memory use reduced to 6 GB ('suggested' limit) and to 4 GB (strict limit). We note that reducing the memory even to 4 GB only moderately increases the processing time. It is worth to note that both KMC 2 and DSK can be run with even lower memory limits, i.e. about 1 GB but it comes at a price of speed drop. For experiments we, however, chose larger settings, as 4-6 GB of RAM seems to fit even low-end machines.

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 as much as 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.

Table 5. Influence of input data format and no./type of drives on the *k*-mers counting times of KMC 2 for *H.sapiens* 2

Drives	$k=28$			$k=55$		
	RAM	Disk	Time	RAM	Disk	Time
Non-gzipped input files						
2 HDDs in RAID 0	12	101	2259	13	70	2640
2 HDDs in RAID 0	4	101	2707	4	70	3471
1 HDD	12	101	2793	13	70	3155
1 HDD	4	101	3274	4	70	3976
1 SSD	12	101	1615	13	70	2038
1 SSD	6	101	1706	13	70	2446
1 SSD	4	101	1843	4	70	2802
Gzipped input files						
2 HDDs in RAID 0	12	101	1868	13	70	2421
2 HDDs in RAID 0	4	101	2237	4	70	3200
1 HDD	12	101	1665	13	70	2112
1 HDD	4	101	2090	4	70	2892
1 SSD	12	101	1217	13	70	1607
1 SSD	7	101	1495	13	70	1909
1 SSD	4	101	1400	4	70	2353

Timings in seconds and RAM/disk consumption in GB.

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

Length	Minimizers			Signatures		
	Avg. in read	No. *k*-mers largest bin	Min. memory	Avg. in read	No. *k*-mers largest bin	Min. memory
$k=28$						
5	6.935	3361	26.5	6.045	1904	18.1
6	7.519	1231	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	3940	62.0	2.477	2257	38.3
6	2.915	1513	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 number 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, $k=28$) and 28.6 GB (minimizers, $k=28$).

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.

To check if the SSD disk, with about 500 MB/s read/write performance, may still be a bottleneck, we ran KMC 2 also in the in-memory mode [rows '(in-mem)' in Tables 2 and 3]. The memory consumption then grows to about the sum of memory and disk use in the standard setting, yet the processing time improves, by about 20% for *G.gallus* and 3% for *M.balbisiana*. This shows that even with the SSD disk, the performance is (somewhat) hampered by I/O operations.

We also measured how the input format (raw, gzipped) and media (one or two HDDs in RAID 0, SSD) affects the performance

Table 7. Impact of (k, x)-mers on bin processing and overall KMC 2 processing, for *G.gallus* and *H.sapiens* 2

x	$k=28$				$k=55$			
	Split time	Sort time	Total time	Sorted fraction	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 2								
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

A 12-GB RAM set, gzipped input. 'Sorted fraction' is the ratio of the number of (k, x)-mers to the number of *k*-mers. 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.

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 [i.e. extracting (k, x)-mers from super *k*-mers] and sorting over longer strings. Still, (k,3)-mers versus 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 Figure 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 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.

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

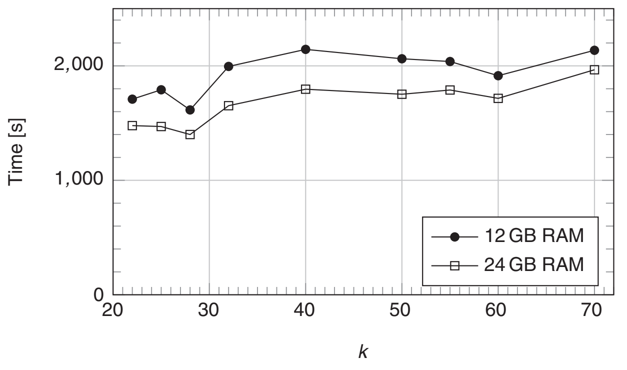

Fig. 4. Dependence of KMC 2 processing time on *k* for *H.sapiens* 2 dataset ($k = 22,25,28,32,40,50,60,70$)

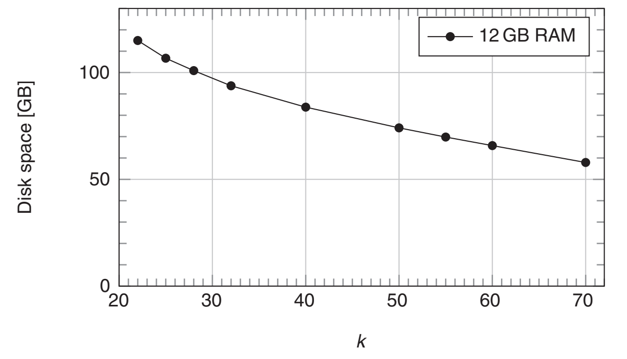

Fig. 5. Dependence of KMC 2 temporary disk usage on *k* for *H.sapiens* 2 dataset

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, Table 1) to about 60% (Sebastião *et al.*, 2012, Table 2) 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.

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 min, 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 gave rather little compared with an SSD (about 5–10% 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.

After our work was ready, we learned about an interesting possibility of using frequency-based minimizers (Chikhi *et al.*, 2014). The idea is to select the (globally) least frequent *m*-mer in a given *k*-mer and it dramatically reduces the memory use in the application of enumerating the maximal simple paths of a de Bruijn graph. In our preliminary experiments freq-based minimizers reduce the memory

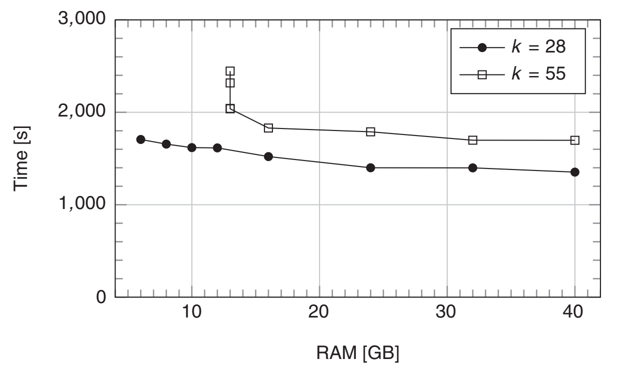

Fig. 6. Dependence of KMC 2 processing time on maximal available RAM and type of disk for *H.sapiens* 2 dataset. There are 4 results for $k = 55$ and 13 GB RAM. These results are for set 6 GB, 8 GB, 10 GB, 12 GB as maximal RAM usage. However, the largest bin enforced to spend at least 13 GB of RAM

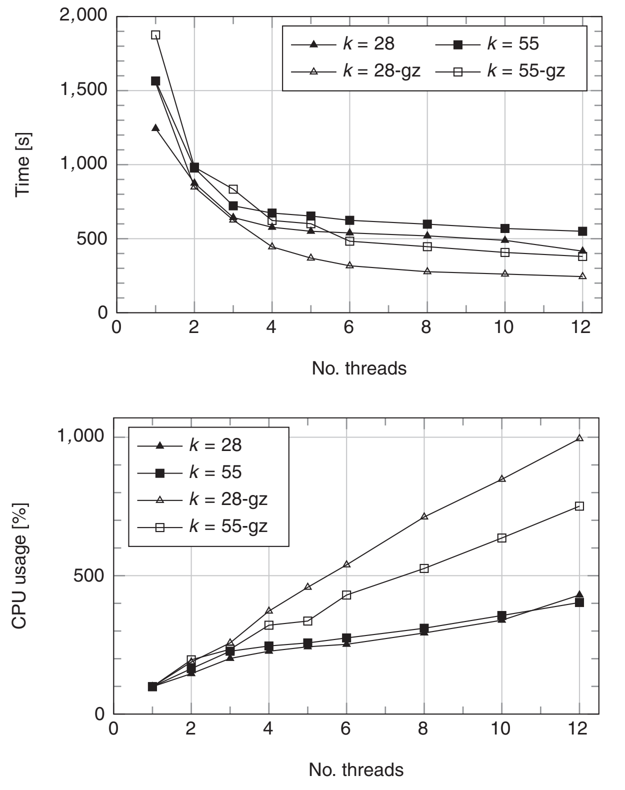

Fig. 7. Dependence of KMC 2 processing time and CPU usage on the set number of threads for *G.gallus* dataset

usage significantly (max bin size is 4 times smaller), but also increase the total I/O. This suggests using the Chikhi *et al*. mechanism optionally, e.g. in low-end machines.

Funding

This work was supported by the Polish National Science Centre under the project DEC-2012/05/B/ST6/03148. This 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'.

Conflict of Interest: none declared.

References

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

Chikhi,R. *et al*. (2014) On the representation of de Bruijn graphs. In: *Research in Computational Molecular Biology (RECOMB)*. Lecture Notes in Computer Science, Vol. 8394, Springer International Publishing, Switzerland, pp. 35–55.

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

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

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

Li,Y. and Yan,X. (2014) MSPKmerCounter: a fast and memory efficient approach for *k*-mer counting. Preprint at http://cs.ucsb.edu/~yangli/paper/bio14_li.pdf

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

Putze,F. *et al*. (2009) Cache-, hash-and space-efficient Bloom filters. *ACM J. Exp. Algor.,* **14**, 4.

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

Roberts,M. *et al*. (2004a) Reducing storage requirements for biological sequence comparison. *Bioinformatics,* **20**, 3363–3369.

Roberts,M. *et al*. (2004b) A preprocessor for shotgun assembly of large genomes. *J. Comput. Biol.,* **11**, 734–752.

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

Schuepbach,T. *et al*. (2013) pfsearchV3: a code acceleration and heuristic to search PROSITE profiles. *Bioinformatics,* **29**, 1215–1217.

Sebastião,N. *et al*. (2012) Implementation and performance analysis of efficient index structures for DNA search algorithms in parallel platforms. Concurrency Comput. Pract. Exp.

Wood,D.E. and Salzberg,S.L. (2014) Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biol.,* **15**, R46.