Kmerind: A Flexible Parallel Library for K-mer Indexing of Biological Sequences on Distributed Memory Systems

Tony Pan, Patrick Flick, Chirag Jain, Yongchao Liu, and Srinivas Aluru

Abstract—Counting and indexing fixed length substrings, or k-mers, in biological sequences is a key step in many bioinformatics tasks including genome alignment and mapping, genome assembly, and error correction. While advances in next generation sequencing technologies have dramatically reduced the cost and improved latency and throughput, few bioinformatics tools can efficiently process the datasets at the current generation rate of 1.8 terabases per 3-day experiment from a single sequencer. We present Kmerind, a high performance parallel k-mer indexing library for distributed memory environments. The Kmerind library provides a set of simple and consistent APIs with sequential semantics and parallel implementations that are designed to be flexible and extensible. Kmerind's k-mer counter performs similarly or better than the best existing k-mer counting tools even on shared memory systems. In a distributed memory environment, Kmerind counts k-mers in a 120 GB sequence read dataset in less than 13 seconds on 1024 Xeon CPU cores, and fully indexes their positions in approximately 17 seconds. Querying for 1 percent of the k-mers in these indices can be completed in 0.23 seconds and 28 seconds, respectively. Kmerind is the first k-mer indexing library for distributed memory environments, and the first extensible library for general k-mer indexing and counting. Kmerind is available at https://github.com/ParBLiSS/kmerind.

Index Terms—k-mer counting, k-mer index, next generation sequencing, distributed computing, parallel computing, MPI, SIMD

1 Introduction

Advances in next-generation sequencing (NGS) technologies have enabled high-throughput sequencing of DNA at dramatically reduced cost, and are continually propelling bioinformatic research into the era of big data. For instance, with respect to whole genome sequencing, one Illumina HiSeq X ten system is able to deliver over 18,000 human genomes with $30\times$ coverage each, in a single year, at a sequencing cost of $<\$1000$ per genome. This high sequencing throughput results in a data volume of about 90 billion base-pairs (bps) per genome and about 1.6 quadrillion bps per year with only one sequencing system. Consequently, big biological sequence analytics have become an increasingly important methodology in biological research and health care, enabled by these cost-affordable and high-throughput sequencing technologies. Projects including The Cancer Genome Atlas [1], The 1,000 Genome Project [2], The 10K Genome Project [3], and even clinical research and health care [4] attest to the ubiquity of NGS and its impacts. The continued adoption of and improvements in sequencing technologies amplify the demand for efficient management and timely processing of biological sequence data using sophisticated bioinformatic algorithms and tools that support data-driven computational tasks [5], [6].

Central to many bioinformatic tasks are k-mer (defined as a length k sequence) counting and indexing, which are widely used in data processing tasks such as sequence search [7], [8], [9], NGS read error correction [10], [11], [12], NGS read alignment [13], [14], [15], and *de novo* assembly [16], [17], [18], [19]. K-mer analysis is also a critical part of applications including sequencing coverage estimation [20], single nucleotide variant identification [10], and metagenomic sequence classification [21], [22]. The broad applicability of k-mer counting and indexing have motivated the development of a diversity of tools providing this capability. Section 2 briefly summarizes the existing works.

In this paper, we present a generic in-memory k-mer indexing and counting library, named *Kmerind*, to address both the performance and data scaling challenges due to or arising from big biological sequence data analysis. In general, the objectives of developing Kmerind include (i) realizing ready scaling of problem size and/or performance with additional hardware resources, (ii) allowing easy configuration and extension with user-specified data types and algorithms, and (iii) offering a consistent set of application programming interfaces (APIs).

In Kmerind, these objectives are directly reflected by our algorithms, APIs, and implementation. Shared-memory computers are inherently limited by computational resources including the number of processor cores and size of memory. In this work, we target distributed-memory environments to support scaling to very large datasets, utilizing very large amounts of memory, and recruiting substantial extra computational resources when performance is paramount. Kmerind classes and functions are written as C++ templates, thus enabling convenient creation of application-specific indices and easy customization of k-mer length,

• *The authors are with the School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: {tpan7, patrick.flick, cjain, yliu860}@gatech.edu, aluru@cc.gatech.edu.*

Manuscript received 2 Dec. 2016; revised 5 July 2017; accepted 20 Sept. 2017. Date of publication 9 Oct. 2017; date of current version 5 Aug. 2019. (Corresponding author: Srinivas Aluru.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCBB.2017.2760829

1545-5963 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

alphabet, and functions associated with index operations. All Kmerind indices are built upon the same set of basic operations, i.e., `insert`, `find`, `count`, and `erase`, with sequential semantics and parallel implementations. Application developers can readily extend the capability of our library by adding new algorithms or optimizing the implementations of existing components.

2 Related Work

K-mer counting and indexing have been extensively investigated in the literature due to their centrality in many bioinformatics algorithms. Most algorithms and software for k-mer analysis target shared memory systems, and operate serially [20], [23], [24], [25], [26], [27] or use multiple threads [28], [29], [30], [31], [32], [33], [34], [35]. To the best of our knowledge, Kmernator [36] is currently the only standalone distributed memory k-mer counter available.

These software target different use cases and provide different interfaces and functionalities. While k-mer indexing software [20], [24], [25] can be used for k-mer counting, the converse may not hold true. A k-mer analysis software may also be designed for a specific pipeline, e.g., assembly, and therefore provides only application-specific query interfaces [33], [36], or only supports off-line queries [28].

Memory is often the primary limitation for shared memory environments. Several specialized approaches have been employed to address this issue, including out-of-core techniques, Bloom filters [37], and succinct data structures such as enhanced suffix arrays [38].

2.1 K-mer Counting

Jellyfish [28] is a popular in-memory k-mer counter and introduces a lock-free hash table to support thread-level concurrent updates. It reduces memory consumption by using a bijective hash function that allows the lower bits of a key to be reconstructed from its hash bucket identifier, thus only the upper bits need to be stored. The memory usage is further minimized by widening the data type only when the k-mer frequency is high. KCMBT [27] is an in-memory k-mer counter that employs cache efficient burst tries.

Out-of-core approach for reducing memory footprint includes the use of disks as external memory. Tools in this group operate with separate partitioning and counting phases. KAnalyze [30] counts k-mers in each input file block and stores the intermediate results, which are aggregated during the counting phase. DSK [29], MSPKmerCounter [26], KMC 2 [33], KMC 3 [35], and Gerbil [34] assign k-mers to on-disk buckets during partitioning, and process each bucket individually during the counting phase.

Probabilistic data structures such as Bloom filters and Count-min Sketch [39] have also been used to reduce memory requirements. BFCounter [23] and Turtle [32] are two examples using Bloom filters. JellyFish 2 [28] also provides an option to support this technique. As Bloom filters can introduce false positives, a second scan of the k-mer counts is necessary. Khmer [31] is a k-mer counter based on Count-min Sketch that, while memory efficient, can overestimate k-mer counts.

Kmerind circumvents single-machine memory limitations by allowing users to recruit additional nodes in a distributed system. In contrast to disk based tools, Kmerind's performance is bound by the faster memory and network rather than disk speed. Unlike techniques involving probabilistic data structures such as Bloom filter [23], [32] or Count-min Hash [31] that exclude singleton k-mers and introduce errors, Kmerind supports exact counting and indexing, and therefore is suitable for a wider spectrum of applications. In addition, Kmerind provides both counting and indexing capabilities via an extensible library API.

2.2 K-mer Position Indexing

Besides k-mer counts, some works further trace the position of each k-mer occurrence through string indexing data structures. Tallymer [20] and Gk-Array [24] are two tools based on enhanced suffix array [38] (an equivalent representation of suffix tree). Välimäki and Rivals [25] extended Gk-Array by proposing a compressed representation based on FM-index [40] that further improves memory efficiency.

Suffix trees and arrays are not well-suited for distributed-memory k-mer counting and indexing. Distributed-memory suffix array construction has been demonstrated previously [41]. However, distributed query processing requires $O(\log(n))$ iterations of sequence comparison, each iteration requiring communication with remote processors. Kmerind instead stores k-mers in data structures that support local associative look-up or comparison-based searches using k-mers as keys.

2.3 Distributed k-mer Indexing

The k-mer counting and indexing tools discussed in Sections 2.1 and 2.2 target shared-memory systems. Kmernator [36] is a hybrid MPI+OpenMP application that implements node- and thread-level master-slave work assignment. However, Kmernator only supports FASTQ files and canonical k-mers.

Distributed-memory assemblers often embed k-mer indexing and counting capability for erroneous k-mer removal and de Brujin graph construction. Examples of such assemblers include ABySS [18], PASHA [42] and HipMer [19]. ABySS uses hash tables, PASHA adopts a combination of hash tables and sorted vectors, while HipMer associates hash tables with Bloom filters. As these k-mer indexing and counting procedures are specialized for assembly only, none of them provides general purpose indexing APIs.

3 Algorithms

Kmerind's algorithms are designed to be efficient in both computation and communication complexities. We leverage bulk-synchronous parallel communication primitives with explicit synchronization semantics to enforce coarse grain synchronization that avoids contention, reduces overhead, and encourages optimization. Specifically, Kmerind algorithms employ primitives defined in version 2 of the Message Passing Interface (MPI) standard [43]. Kmerind does not use multithreading as thread-safety incurs additional overheads for this highly data parallel task. To minimize costly file system access, Kmerind indices are memory resident for the duration of their use.

The number of processors used is denoted as p. Communication complexity is described in terms of latency τ, bandwidth $1/\mu$, and message size m. Point-to-point communication, e.g., `MPI_Send`, has complexity $O(\tau + \mu m)$, while collective communication, e.g., `MPI_Alltoallv`, has $O(\tau \log(p) + \mu m \log(p))$ assuming hypercubic permutation based implementation.

The number of occurrences of a k-mer in a dataset is referred to as its *count* or *frequency*. We use the term *distinct*

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

to describe k-mers with different character sequences, in contrast to *unique* k-mers whose frequency is 1. The set of all indexed k-mers is denoted by N, whose distinct k-mer subset is denoted by U. The set of input k-mers for an index operation is denoted by M. The subscripted versions N_i, U_i, and M_i denote the corresponding subsets on processor i. The size of a set is represented by $|\cdot|$. The average global and per-processor k-mer frequencies are denoted by $r = |N|/|U|$ and $r_i = |N_i|/|U_i|$.

References to lines in algorithms use the formats "Ax:y", "Ax:y, z" and "Ax:y–z" for efficiency, where x is the algorithm number and y and z are the line numbers.

3.1 Distributed k-mer Parsing

Kmerind supports parallel file reading and k-mer parsing. Currently, FASTQ format, which is primarily used for storing NGS sequence reads, and FASTA format, which is used for sequence reads as well as whole genomes, are supported. We denote the sequence file as F.

Parallel file reading and k-mer parsing proceeds in 3 steps, *file partitioning*, *sequence segmentation*, and *k-mer generation*, as shown in Fig. 1 and Algorithm 1. Parallelization circumvents single machine limits. Each step maintains load balance across processors and linear time complexity.

During *file partitioning* (A1:2–3), F is divided into approximately equal partitions of $|F|/p$ bytes and loaded in parallel into main memory as a character array S. The *sequence segmentation* step (A1:5,6,15) employs linear-time, format-specific logic to identify in S individual sequences, from which k-mers, positions, and other data are extracted. For each segmented sequence, the *k-mer generation* step (A1:8-14) extracts k-mers via a sliding window and stores the results in an array to be used as input for the distributed index operations. When an unknown character is encountered, one of three strategies is used: discard the sequence; discard the k-mer; or replace the unknown with a known character such as "N".

Sequence segmentation algorithms for the FASTQ and FASTA formats are outlined in the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TCBB.2017.2760829. Briefly, for a FASTQ file the partition boundaries are adjusted to begin and end at sequence boundaries.

Algorithm 1. Distributed k-mer Parsing

```
 1: function ParseKmers(F, rank, p)
 2:   ⟨start, end⟩ ← |F|/p × ⟨rank, rank + 1⟩
 3:   S ← read_file(F, start, end)
 4:   kmers ← empty array
 5:   InitSequenceSegmention(S, rank, p)
 6:   seq ← GetNextSequence(S, start, end)
 7:   while seq ≠ empty do
 8:     for j ← 0, |seq| do
 9:       kmer ← (kmer ≪ 1)
10:       kmer ← kmer.append(seq[j])
11:       if j ≥ k − 1 then
12:         kmers.append(kmer)
13:       end if
14:     end for
15:     seq ← GetNextSequence(S, seq.end, end)
16:   end while
17:   return kmers
18: end function
```

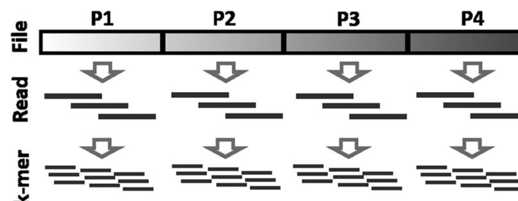


Fig. 1. Parallel sequence file reading in distributed environment.

Sequences in S are then identified iteratively leveraging FASTQ format regularity. For FASTA files, the starting and ending positions of all sequences are computed and stored, since a sequence may span multiple processors and computing the positions requires global communication. Subsequent segmentation uses the stored positions.

3.2 Distributed Indices: Overview

Kmerind's distributed k-mer indices are modeled as either hash maps or sorted arrays of k-mers and associated data. In both cases the k-mer space is partitioned amongst the processors so that an input k-mer for an index operation is deterministically assigned to and processed by one and only one processor. We also considered look-up tables and suffix arrays. While a distributed look-up table can enumerate the entire k-mer space as a 4^k array for small k, non-uniformity in k-mer distribution of the genome or read file can translate to computational load imbalance. Suffix arrays and trees, while flexible and memory efficient, are not well suited for distributed-memory queries as stated in Section 2.

Kmerind's distributed *hashed* index is designed as a two-level hash map. The upper level hash function maps k-mers to processor ranks uniquely and deterministically, while the lower level consists of a local hash map for storing k-mers and associated data. Hash function for each level is user definable and should be chosen to produce (1) uniformly distributed hash values to ensure load balance across processors and minimize hash collisions within local hash maps, and (2) uncorrelated upper and lower level hash values to avoid clumping, where k-mers mapping to the same processors are assigned to the same map buckets.

Kmerind's distributed *sorted* index stores a k-mer and its associated data as a tuple in a distributed, sorted array. A size $p - 1$ array of *splitter* k-mers, replicated on all processors, maps a k-mer to its assigned processor via binary search.

User preference and application needs dictate the choice of hashed versus sorted indices. Hashed index allows expected $O(1)$ time access to the k-mers, at the expense of extra space for empty map buckets and hash map overheads. Sorted index carries an additional logarithmic factor in time but requires only as much space as the k-mer data and can be partitioned equally across processors for non-uniformly distributed k-mer set. A sorted index may facilitate integration with an application's native data structures and simplify communication by avoiding extraneous copies.

Kmerind indices are further classified as *uni-* and *multi-*indices. Uni-indices store one instance of each k-mer and associated data, for example for k-mer counting. Multi-indices store multiple instances of each k-mer, for example to index k-mer's positions.

Kmerind defined 4 basic operations that are categorized as *update* or *query* operations. *Update* operations include `insert` and `erase`, where communication is one-way only. *Query* operations, on the other hand, require round-trip communication and include `count` and `find` operations.

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

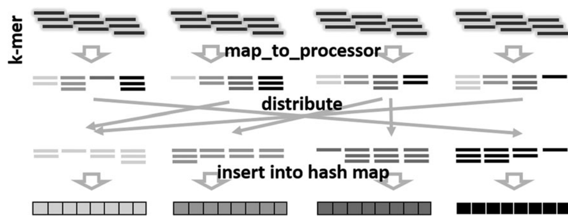


Fig. 2. Inserting k-mers into the Kmerind's distributed index.

We expand the set of notations with R_i, which denotes the results of a *query* operation. Subscript "r" before a variable, e.g., ${}_rM_i$, denotes the remote copy of the variable, M_i, after a collective communication such as `MPI_Alltoallv`. Apostrophe, in M_i' for example, indicates that a variable has been locally permuted for a purpose such as bucketing prior to collective communication. Note that permutation does not change the size of the array. The local data store is denoted by C, with $|C| = |N_i|$. We further assume p is much smaller than $|N_i|$, $|M_i|$, and $|R_i|$.

3.3 Distributed Hashed Index

For each operation in a distributed hashed index, the input data is first assigned (`map_to_processor`) then communicated to the target processor (`distribute`). The communicated data is then processed locally on each processor, with results optionally communicated back to the source processor. Fig. 2 illustrates this process for insertion.

3.3.1 Data Movement

Input data movement for an operation of a distributed hashed index begins with the `map_to_processor` operation outlined in A2, where `mapper` is the upper level hash function. The k-mers are first assigned to target processors using the upper level hash function (A2:4-7). Both the assignments and the number of k-mers in each bucket are saved simultaneously. The k-mers in the input array are then stably permuted (A2:14-17) so that k-mers for the same processor occupy contiguous memory as required by the communication primitives. The bucket assignment array is converted into a *source-to-target* position mapping array (A2:8-13). The position mapping array is used during permutation, and for inverse permutation if input data in the original order is needed.

Algorithm 2. map_to_processor

1: **function** MAP_TO_PROCESSOR(M_i, `mapper`, p)
2: $map \leftarrow$ array of size $|M_i|$
3: $counts \leftarrow$ array of size p
4: **for** $i \leftarrow 0, (|M_i| - 1)$ **do**
5: $map[i] \leftarrow$ `mapper`$(M_i[i])$ mod p
6: $counts[map[i]] \leftarrow counts[map[i]] + 1$
7: **end for**
8: offsets $\leftarrow$ exclusive_prefix_sum($counts$)
9: **for** $i \leftarrow 0, (|map| - 1)$ **do**
10: $rank \leftarrow map[i]$
11: $map[i] \leftarrow$ offsets[$rank$]
12: offsets[$rank$] $\leftarrow$ offsets[$rank$] +1
13: **end for**
14: $M_i' \leftarrow$ array of size $|M_i|$
15: **for** $i \leftarrow 0, (|M_i| - 1)$ **do**
16: $M_i'[map[i]] \leftarrow M_i[i]$
17: **end for**
18: return $\langle M_i', counts, map \rangle$
19: **end function**

Once the k-mers have been assigned and arranged by target processors, the `distribute` function sends k-mers and associated data to target processors using the collective personalized communication primitive, `MPI_Alltoallv`. A corresponding `inverse_distribute` function is defined to move k-mers and associated data back to their source processors, for example to return *query* operation results. This is accomplished by first permuting the *counts* array using `MPI_Alltoall` to get the element counts in the received buckets, then using the permuted *counts* and the k-mer array as argument for `distribute`.

COMPLEXITY ANALYSIS: The overall time complexity for `map_to_processor` is linear in the size of the input $O(|M_i|)$, as each of the 3 `for` loops requires constant time per iteration over $|M_i|$ iterations. The local *exclusive prefix sum* requires $O(p)$ time. The total time and space complexities are $O(|M_i|)$, assuming $p \ll |M_i|$. No communication is incurred during this operation.

The complexity of the `distribute` function depends directly on the space and time complexities of MPI operations, $O(\tau \log(p) + \mu |M_i| \log(p))$ time and $O(|M_i| + |{}_rM_i|)$ space, linear in the size of the input and output.

3.3.2 Distributed Hashed Uni-Index

Kmerind's distributed hashed uni-index allows a single value to be associated with each k-mer, and supports *update* operations `insert` and `erase`, as well as *query* operations `count` and `find`. For each *query* k-mer, at most one result value is returned. Therefore, $|M_i| = |R_i|$ and $|{}_rM_i| = |{}_rR_i|$

The algorithms for `insert` and `count` index operations are shown in A3 and A4, respectively, where M_i is an array of query k-mers on processor i, C is the local container, and R_i contains the query results for processor i. In each algorithm, the input is assigned and distributed to the target processors using `map_to_processor` and `distribute` operations as described in Section 3.3.1. The target processors then process the received k-mers and data locally (A3:5, A4:5). The `count` algorithm returns the count results to the source processors via collective communication using the `inverse_distribute` operation.

Algorithm 3. Distributed Hashed Index Insert

1: **procedure** INSERT(M_i, C, `mapper`, p)
2: $\langle M_i', counts \rangle \leftarrow$ map_to_processor(M_i, `mapper`, p)
3: ${}_rM_i \leftarrow$ distribute(M_i', $counts$)
4: **for** $x \in {}_rM_i$ **do**
5: $v \leftarrow C$.`insert`(x)
6: **end for**
7: **end procedure**

Algorithm 4. Distributed Hashed Index Count

1: **procedure** COUNT(M_i, C, `mapper`, p, R_i)
2: $\langle M_i', counts \rangle \leftarrow$ map_to_processor(M_i, `mapper`, p)
3: ${}_rM_i \leftarrow$ distribute(M_i', $counts$)
4: **for** $x \in {}_rM_i$ **do**
5: $v \leftarrow$ C.`count`(x)
6: ${}_rR_i$.append($\langle x, v \rangle$)
7: **end for**
8: $R_i \leftarrow$ inverse_distribute(${}_rR_i$, $counts$)
9: **end procedure**

The algorithms for `erase` and `find` are essentially identical, except the local hash map operations at A3:3 and A4:4

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

are replaced with C.erase and C.find, respectively. Duplicates in input may be removed before invoking the hashed index operations to reduce subsequent communication volume and computational load.

COMPLEXITY ANALYSIS: We assume appropriate hash functions were chosen so that data is distributed as uniformly as possible, $|M_i| \approx |_rM_i|$, and that the local hash map has expected $O(1)$ access time per k-mer.

Update operations insert and erase requires $O(|M_i|)$ space and time complexity for map_to_processor (A3:2, A4:2), $O(|M_i| + |_rM_i|)$ space and $O(\tau \log(p) + \mu|M_i|O(\tau \log(p) + \mu|M_i| \log(p))\log(p))$ time for distribute (A3:3, A4:3), and $O(|_rM_i|)$ time and space for local hash map insertion and erasure (A3:5, A4:5). *Update* operations have overall complexity of $O(|M_i| + |_rM_i|)$ in space, and $O(\tau \log(p) + \mu|M_i| \log(p))$ in communication time and $O(|M_i| + |_rM_i|)$ in computation time.

The *query* operations count and find differ from the *update* operations by their results handling, which requires $O(|_rR_i| + |R_i|)$ space, and $O(\tau \log(p) + \mu|_rR_i| \log(p))$ communication time, and $O(|_rR_i| + |R_i|)$ computation time. Since $|_rR_i| = |_rM_i|$ and $|R_i| = |M_i|$ for uni-indices, the overall space and computation time complexity of the *query* operations remains the same as those for the *update* operations, while the communication time complexity becomes $O(\tau \log(p) + \mu(|M_i| + |_rM_i|) \log(p))$.

Assuming equal input and distributed data partitioning, $|M_i| = |_rM_i| = |M|/p$, the *update* and *query* operations then have space and computation time complexity linear in the size of the input, $O(|M|/p)$, and communication time complexity of $O(\tau \log(p) + \mu|M|/p \log(p))$.

3.3.3 Distributed Hashed Multi-Index

Kmerind's hashed multi-index uses a local hashed multi-map to associate multiple values to each k-mer. The local hashed multi-map implementation can affect the per-element access time complexities, however. Kmerind's choice of local hash multi-map is described in Section 4.2. Here we assume that time complexity is expectedly $O(1)$ for each local multi-map access.

Kmerind's hashed multi-index and uni-index share the same algorithm for *update* operations, which processes each input k-mer regardless of multiplicity in the associated data for each k-mer. For count operation, since exactly one count response is generated for each query k-mer, the algorithm for uni-index's count operation is adopted for the multi-index count operation.

Unlike the uni-index find operation, however, $|R_i| \neq |M_i|$ and $|_rR_i| \neq |_rM_i|$ for the multi-index find operation. Furthermore, $_rM_i$ may contain replicated query k-mers from different processors.

We assume equal partition for the distinct k-mers U in the indexed k-mers, $|U_i| = |U|/p$. We further assume that the query k-mers M are sampled from the same distribution as N and equal partitioning of M. Then M_j on processor j has expected size $|M_j| = r|U|/p$, and the distributed input k-mer set on processor i has expected size $|_rM_i| = r_i|U|/p$, which implies that the intermediate results have size $|_rR_i| = r_i^2|U|/p$. Assuming each input subset sent from processor j to i contains $r_i|U|/(p^2)$ k-mers, then the final results have size $\Sigma_{i=0}^{p-1} r_i^2|U|/(p^2) = (|U|/p)(\Sigma_{i=0}^{p-1} r_i^2)/p$.

The second order dependence of $|_rR_i|$ on r_i implies that non-uniformity in frequency distribution can quickly cause load imbalance in computation, memory usage, and communication for the uni-index *query* algorithm A4.

Algorithm 5. Distributed Hashed Multi-Index find

```
 1: procedure FIND(M_i, C, mapper, p, r_i, R_i)
 2:   ⟨M'_i, counts⟩ ← map_to_processor(M_i, mapper, p)
 3:   _rM_i ← distribute(M'_i, counts)
 4:   B_i ← empty array of size p
 5:   for j ← 0, (p − 1) do
 6:     _rM_ij ← subset of _rM_i from processor j
 7:     for x ∈_r M_ij do
 8:       B_i[j] ← B_i[j] + C.count(x)
 9:     end for
10:   end for
11:   B_i ← inverse_distribute(B_i, counts)
12:   c ← sum(B_i)
13:   R_i ← empty array of size c
14:   for j ← 0, (p − 1) do
15:     k ← (i + j) mod p
16:     _rM_ik ← subset of _rM_i from processor k
17:     T ← empty array
18:     for x ∈_r M_ik do
19:       T.append( C.find(x) )
20:     end for
21:     T ← Send(T, k)
22:     R_i.append(T)
23:   end for
24: end procedure
```

Instead, Kmerind's position index's *find* operation uses Algorithm 5 that amortizes the space and time requirements over p iterations. During each iteration, the query k-mers from one source processor are processed. The query k-mers are first assigned and distributed (A5:2-3). The total number of result tuples are counted and returned to the source processor (A5:4-11) so that the result array R_i can be allocated (A5:12-13). We then iterate over each query k-mer subsets $_rM_{ik}$ by processor rank k (A5:14-23), finding all results for a subset (A5:19) and sending the subset result immediately (A5:21) before processing the next subset. Non-blocking point-to-point communication (MPI_Isend and MPI_Irecv) is used, which allows communication to overlap query processing computations.

COMPLEXITY ANALYSIS: Distributed hashed multi-indices have identical complexities for the insert, erase, and count operations as those for the uni-indices (Section 3.3.2).

The find algorithm for the multi-index aims to minimize the intermediate memory requirement $|_rR_i| = r_i^2|U|/p$. Processing the query k-mer subsets of $_rM_i$ iteratively requires at most $O(r_i \max_k(|_rM_{ik}|))$ space due to buffer reuse. Across all iterations, the computation and communication time complexities are $O(r_i|_rM_i|)$ and $O(\tau p + \mu r_i|_rM_i|)$ respectively. The additional counting step has the same computation time as query processing, therefore does not affect the overall complexity.

The overall complexity of the hashed multi-index find operation is dominated by the subset query processing iterations, $O(\tau p + \mu|M_i| \log(p) + \mu r_i|_rM_i|)$ for communication, $O(|M_i| + r_i|_rM_i|)$ for computation, and $O(|M_i| + |_rM_i| + |R_i| + r_i \max_k(|_rM_{ik}|))$ for space. The algorithm avoids the quadratic intermediate result space requirement for highly repeated k-mers from $O(r_i|_rM_i|)$ to $O(r_i \max_k(|_rM_{ik}|))$. Communication message sizes are likely more balanced, and the

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

bandwidth term is reduced from $O(\mu(|M_i| + r_i|_rM_i|)\log(p))$ for collective communication to $O(\mu(|M_i|\log(p) + r_i|_rM_i|))$ at the expense of increased latency term, τp.

3.4 Distributed Sorted Index

Kmerind's distributed sorted indices store $<k$-mer, value$>$ tuples in a sorted array using k-mer as key. A sorted array has a strict ordering by k-mer values, and tuples with identical keys, such as those in multi-index, are arranged contiguously in the array. The sorted array is partitioned as equally as possible across all processors. We further require that tuples with identical keys reside on the same processor.

3.4.1 Data Movement

Leveraging these properties and requirements, and after an array has been sorted, we can establish a surjective mapping from query k-mers to partitions of the sorted array, each residing on a processor. A simple approach adopted by Kmerind is to use the last k-mer from each partition as a *splitter*. Through binary search in the array of *splitters* of size $p-1$, a query k-mer can be uniquely and deterministically assigned to a processor.

We adopt hashed index's `map_to_processor` algorithm (A2), with the exception that line 5 is replaced with a binary search for the query k-mer's insertion position in the *splitter* array. The insertion position corresponds to the processor rank to which the k-mer should be sent.

Subsequent to mapping, the query k-mers can be sent to the target processors via the `distribute` and `inverse_distribute` operations from Section 3.3.1.

COMPLEXITY ANALYSIS: The `map_to_processor` operation uses a binary search in the *splitter* array for each k-mer. The *splitter* array is replicated on each processor and requires p space. Overall time complexity is $O(|M_i|\log(p))$, with $\log(p)$ from the binary search. The `distribute` algorithm has identical time and space complexity as those for the hashed index's `distribute` operation.

Algorithm 6. Local Erase for Sorted Index

1: **procedure** ERASE($_rM_i$, C)
2: out_pos $\leftarrow 0$
3: start $\leftarrow 0$
4: end $\leftarrow 0$
5: $_rM_i \leftarrow$ sort($_rM_i$)
6: **for** $x \in {_rM_i}$ **do**
7: end $\leftarrow$ lower_bound_pos($C[start \ldots |C|], x$)
8: **for** $i \leftarrow$ start, (end - 1) **do**
9: C[out_pos] $\leftarrow$ C[i]
10: out_pos $\leftarrow$ out_pos + 1
11: **end for**
12: start $\leftarrow$ upper_bound_pos($C[end \ldots |C|], x$)
13: **end for**
14: **for** $i \leftarrow$ start, $(|C| - 1)$ **do**
15: C[out_pos] $\leftarrow$ C[i]
16: out_pos $\leftarrow$ out_pos + 1
17: **end for**
18: **end procedure**

3.4.2 Distributed Sorted Uni- and Multi-Indices

Data movement in Section 3.4.1 depends on the presence of the *splitter* array, which is constructed during or after parallel sorting. The `insert` operation for distributed sorted uni-index and multi-index employs parallel sample sort [44] with regular sampling to sort the input k-mer array and produce the *splitter* array concurrently.

The `erase`, `count` and `find` operations for both the sorted uni- and multi-indices employ the same algorithms as those for the hashed indices: A3, A4, and A5. The hashing `map_to_processor` operations (A3:2, A4:2, A5:2) are replaced with the binary search version described in Section 3.4.1. The local hash map `erase` (A3:4-6), `count` (A4:4-7), and `find` (A5:18-20) operations are likewise replaced with their sorted array counterparts.

During the local `erase` operation (A6), the query k-mers $_rM_i$ are first sorted so that binary search ranges in the indexed array can be reduced successively. For each query k-mer, the matching range in the sorted array is identified (A6:7,12) then overwritten in the next iteration with the array elements *between* successive matching ranges (A6:8-11, A6:14-17). After all query k-mers are processed, the remaining non-matching array elements are moved forward. The sorted array size is thus reduced.

The local `count` algorithm (A7) similarly sorts the query k-mers first. For each k-mer in the query set, the matching range is computed via 2 binary searches (A7:7-8), then the count is computed from the range (A7:9). The local `find` algorithm differs only in that elements in the round range are copied to R_i instead of computing the count (A7:9).

COMPLEXITY ANALYSIS: For distributed sorted indices, k-mer frequency in the sorted array affects the algorithm trivially and the complexity is increased by a factor of r_i for the terms related to the output data.

The `insert` operation for Kmerind's distributed sorted indices has time complexity equal to that of parallel sample sort [44]. The computation time is dominated by local sorting $O(|M_i|\log(|M_i|))$ assuming total sample size is negligible compared to input data size, $p^2 \ll |M_i|$, while communication complexity is $O(\tau\log(p) + \mu(p + |M_i|)\log(p))$. Space required is primarily for communication buffers thus linear in input size, $O(|M_i|)$.

Algorithm 7. Local Count for Sorted Multi-Index

1: **procedure** COUNT($_rM_i$, C, R_i)
2: start $\leftarrow 0$
3: end $\leftarrow 0$
4: $R_i \leftarrow$ empty array
5: $_rM_i \leftarrow$ sort($_rM_i$)
6: **for** $x \in {_rM_i}$ **do**
7: start $\leftarrow$ lower_bound_pos(C[end, $|_rM_i|$], x)
8: end $\leftarrow$ upper_bound_pos(C[start, $|_rM_i|$], x)
9: R_i.append($\langle x$, end - start $\rangle$)
10: **end for**
11: **end procedure**

The `erase`, `count`, and `find` operations have identical space and communication complexities as those for the hashed uni- and multi-indices, bound by the total input and output sizes. As the local query processing algorithms are specific to sorted arrays, the computation complexities involves a scaling factor from searching the index data. Here we assume that binary search is used with per-query complexity of $O(\log(N_i))$.

For the `erase` operation, the computation complexity is $O(|M_i|\log(p) + |_rM_i|\log(|N_i|) + |N_i|)$. The first term is for assigning query k-mer to processors, the second for searching for matching k-mers, while the third is for moving

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

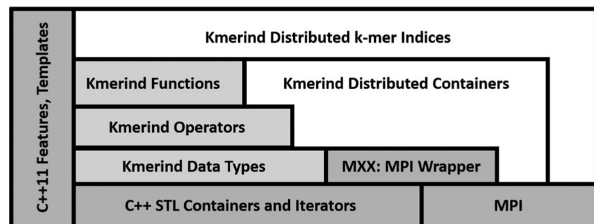


Fig. 3. Kmerind library's tiered architecture.

non-matching k-mers. The `count` operation has similar computation complexity, with the last term being $|_rM_i|$ for computing the size of the matching range for each query k-mer. For for both uni- and multi-indices, the last term for the find operation is $r_i|_rM_i|$, due to copying of all elements in the matching range.

4 Implementation

We designed and implemented Kmerind as a distributed memory parallel library based on the objectives listed in Section 1. Kmerind is a header only C++ library with a tiered architecture (Fig. 3). It leverages C++ 11 language features, Standard Template Library (STL) containers and algorithms, and MPI and the *mxx* [45] MPI wrapper library. Each tier defines templated functions and class interfaces as well as default implementations to allow functionality by composition and extension by specialization and inheritance, thus providing Kmerind's flexibility and extensibility.

The *Data Types* layer defines alphabet and k-mer types and associated operations such as k-mer reverse complement. The *Operators* layer defines transformations that facilitate sequence segmentation and k-mer parsing. The parallel file reader and k-mer generator in the *Functions* layers use these operators to parse files of different formats and generating $\langle k\text{-mer, value}\rangle$ pairs of various types.

The *Distributed k-mer Indices* layer contains k-mer indices, which are implemented as light-weight wrappers for *Distributed Containers*. The distributed containers implement algorithms described in Sections 3.3 and 3.4, and use local hash maps or sorted arrays for local storage. Additionally, Kmerind provides maps that perform reduction on insertion, an example being a counting map.

Where possible, the API presents sequential semantics for simplicity, and encapsulates distributed memory implementation details.

4.1 K-mer Representation

In Kmerind, k-mers are specified via length k and alphabet Σ. Kmerind allows arbitrary k values, including even values. Three primary alphabets, DNA, DNA5, IUPAC DNA, and their RNA equivalents, have been provided. The DNA alphabet consists of $\{A, C, G, T\}$, while DNA5 adds N to denote an unknown nucleotide. IUPAC DNA uses 16 characters to represent the power set of the four DNA nucleotides, e.g., K represents either G or T. Each alphabet also defines the complement mapping for its nucleotides and minimal bit encoding for each character.

As DNA is double stranded, each k-mer x has a reverse complement $\bar{x}$ on the opposite strand, and a canonical representative, $\tilde{x}$, defined as the smaller of x and $\bar{x}$.

Kmerind k-mers are compressed using character encodings with the minimal number of bits $b = \lceil \log(|\Sigma|) \rceil$. For DNA, DNA5, and IUPAC DNA, the bit lengths are 2, 3, and 4 respectively. A k-mer is represented by kb bits in an array of machine words, with unused bits in the most significant positions. Operations on k-mers have complexities that depend linearly on k and the machine word size.

TABLE 1
SIMD-Friendly Bit Encoding for DNA, DNA5, and IUPAC DNA Characters and Corresponding Character Complement Method

	Character		Complement		Complement Method
	Char	Bits	Char	Bits	
DNA	A	00	T	11	negate
	C	01	G	10	
DNA5	gap	000	gap	000	bit reverse
	A	001	T	100	
	C	011	G	110	
	N	111	N	111	
IUPAC	gap	0000	gap	0000	bit reverse
	A	0001	T	1000	
	C	0010	G	0100	
	R (A,G)	0101	Y (C,T)	1010	
	...		...		
	N	1111	N	1111	

For IUPAC DNA alphabet, not all characters are shown.

Rather than explicitly model double stranded k-mers, Kmerind accounts for the double stranded nature in the indexing operations. Kmerind indices can manage and query each k-mer as-is (single strand mode), convert each k-mer to canonical (canonical mode), or store k-mer as is but accept x and $\bar{x}$ as equivalent for queries (bimolecule mode). In bimolecule mode, an index's hash and comparison functions compute $\tilde{x}$ on demand. The performance of k-mer reverse complement operation `revcomp` is critical and has been vectorized using Single Instruction Multiple Data (SIMD) hardware instructions and SIMD Within A Register (SWAR) [46] patterns where only x86 instructions are used.

The `revcomp` operation proceeds in two conceptual phases: character order reversal and character complement. To reverse the order of characters, each word in a k-mer is byte-reversed, and each byte is then character-reversed. Machine words in a k-mer are processed in linear order.

SIMD byte reversal uses the SSSE 3 or AVX 2 `pshufb` instruction with a look up table of reversed positions, while SWAR byte reversal uses the x86 `bswap` instruction. SIMD character reversal within a byte again uses `pshufb` but with a look up table of reversed characters. SWAR character reversal employs bitwise *mask-shift-or* pattern to swap blocks of characters within each bytes over $\log(8/b)$ iterations.

To accelerate character complement, the bit encoding of characters as defined by Σ are chosen so that the complement of a character can be computed via simple vectorizable functions. For the DNA5 and IUPAC DNA, the complement function is *bit reversal*, while for the DNA alphabet, *bitwise negation* is used. Examples are given in Table 1.

For encodings where complements are computable via bit-reversal, the character reversal mechanism is extended to compute reverse complement in one step. This approach allows DNA5 `revcomp` to be implemented in the same way and with similar running time as that for IUPAC DNA, despite the lack of byte-alignment.

4.2 Local Hash Table

Kmerind's distributed hash map implementation allows different local hash map implementations to be used.

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

TABLE 2
Strategies for Choosing k-Mers as the *Empty* Key for *DHM*

Condition	Strategy	example
▹ DNA5	via unused encoding: 010	000 000 010
▹▹ Has unused bits	set highest unused bits	10 11 10 01
▹▹▹ Is canonical index	use *un*-canonical k-mer	TTT
▹▹▹▹ all others	split k-mer space lower k-mer space map higher k-mer space map	 TTT AAA

Conditions listed are checked successively row by row. If a condition is met, the strategy listed on that row is used. Examples shown are 3-mers in ASCII or binary encoding.

Kmerind incorporates Google SparseHash's Dense Hash Map (referred to as *DHM*) [47] as the default local hash table due to its performance. *DHM* uses open addressing with quadratic reprobing, thus requiring 2 dedicated keys to identify *empty* and *deleted* hash table slots. The choice of these keys depends on k-mer parameters and the strand mode of the index as defined in Section 4.1. Table 2 summarizes the decision tree for selecting the strategy to generate the *empty* key for *DHM*. *Deleted* key selection is similar.

The general approach for choosing the *empty* and *deleted* keys is to identify invalid bit patterns, such as unused character bit encoding in DNA5 or available padding bits in the most significant word of a k-mer. For canonical-mode indices, *un*-canonical k-mers can serve as keys. Failing both, the k-mer space can be partitioned between two DHMs with k-mers from the opposite partition as keys. This last approach is extensible to the distributed memory environment, where a processor's k-mer space partition can provide keys for the next processor's DHM instance.

We extended *DHM* into a multimap in order to support distributed multi-maps. *Dense Hash MultiMap*, or *DHMM*, allows multiple values per k-mer key through indirection to secondary arrays. *DHMM* stores singleton k-mers in one array, referred to as *SA*, and replicated k-mers in an array of arrays, referred to as *MA*, where each inner array contains all values associated to a particular k-mer. An internal *DHM* stores k-mer and array position pairs, with the position value sign bit selecting *SA* or *MA*. Using positions instead of pointers or iterators allows *SA* and *MA* to dynamically resize without costly internal *DHM* rebuilds and improves cache utilization. Separate arrays for unique and replicated k-mers minimizes the number of memory allocations for inner arrays of *MA*.

COMPLEXITY ANALYSIS: *DHM* has amortized $O(1)$ `insert` and expected $O(1)$ `find`, `count` and `erase` time complexities as designed and implemented. Kmerind pre-allocates the local hash table if possible to reduce memory allocation cost.

Insertion in *DHMM* requires amortized $O(1)$ time as the *SA* and *MA* array as well as the internal *DHM* may resize as needed. Counting requires constant time as the counts for singleton k-mers are always 1, while the counts for repeated k-mers are the sizes of the corresponding inner arrays. Deletion requires constant time since only the internal *DHM* needs to be modified to mark an entry as deleted. To retrieve all values mapped to a k-mer, *DHMM* requires time linear in the size of the output, on average $O(r)$.

4.3 K-mer Count and Position Indices

Kmerind provides default count and position index implementations. The count index specifies $\langle k\text{-mer, count}\rangle$ as index elements, $k\text{-mer} \in U$. The count index is implemented using either the distributed hashed or sorted reduction map with $+$ operator over the *count* field. The default position index specifies $\langle k\text{-mer, position}\rangle$ as index elements, and is implemented using the distributed hashed or sorted multimap. In both cases, the user can specify k, Σ, and index mode (canonical, single, bimolecule). In the case where hashed map is used, the upper and lower hash functions can be specified. Experiments use the default implementations and the discussions refer to count and position indices directly.

We note that while de-duplicating the query k-mers theoretically helps to reduce communication, computation, and memory costs, its practical utility is limited for `insert`, `erase`, and `count` operations. The expected number of replicated query k-mers on a processor, $(1/|U|)(|N|/p) = r/p$, decreases with increasing p. For a typical whole genome sequencing dataset with $30\times$ coverage and $p = 32$, we expect that most k-mers are locally distinct. De-duplication is therefore only implemented for the `find` operation, where frugal memory usage is more critical.

5 EXPERIMENTAL RESULTS

We examined the performance of Kmerind and compare it to those of a select subset of existing k-mer indexing tools. Datasets used for the experiments are summarized in Table 3, and referenced by "Ids" in subsequent discussions.

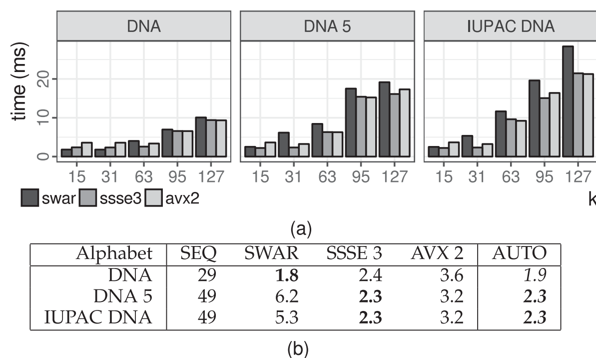
Sequential and multi-threaded tests were conducted on the CompBio system at Georgia Institute of Technology.

TABLE 3
Experimental Datasets Used for All Evaluations

Id	Organism	Genome Size (Mbases)	File Format	File Count	File Size (Gbytes)	Sequence Count	Average Read Length	Source	Accession
R1	H. sapiens	2,991	FASTQ	1	6.3	23,861,612	101	1,000 Genome HG00096, NCBI	SRR077487 forward only
R2	F. vesca [48]	214	FASTQ	11	14.1	12,803,137	352	NCBI	SRA020125
R3	G. gallus	1,230	FASTQ	12	115.9	347,395,606	100	NCBI	SRA030220
R4	H. sapiens	2,991	FASTQ	48	424.5	1,339,740,542	101	NCBI	ERA015743
G1	H. sapiens	2,991	FASTA	1	2.9	84	–	1,000 Genome reference GRCh37	–
G2	P. abies [49]	20,000	FASTA	1	12.4	10,253,694	–	Congenie.org	–
M1				1	7.6	33,195,888			
M2				1	15.2	66,391,776		IOWA Continuous	
M3	metagenome	–	FASTQ	1	30.4	132,783,552	101	Corn Soil (Project 402461),	–
M4				1	60.8	265,567,104		Joint Genome Institute	
M5				1	121.6	531,134,208			

Where applicable, accession numbers for NCBI are provided.

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap



(a)

Alphabet	SEQ	SWAR	SSSE 3	AVX 2	AUTO
DNA	29	**1.8**	2.4	3.6	*1.9*
DNA 5	49	6.2	**2.3**	3.2	**2.3**
IUPAC DNA	49	5.3	**2.3**	3.2	**2.3**

(b)

Fig. 4. Time in milliseconds to reverse-complement one million k-mers of varied alphabets using SWAR, SSSE 3, or AVX 2 instructions for different k (a) and 31-mers (b).

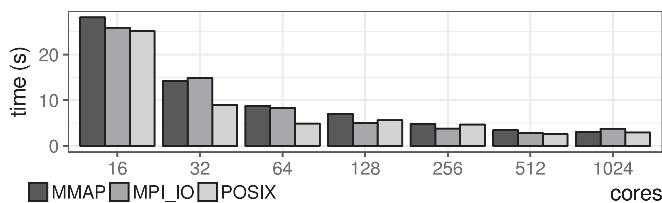


Fig. 5. Time in seconds to read and parse the $\mathbf{R1}$ dataset from disk into memory via MPI-IO, POSIX, and memory mapping, using varying number of cores. The x-axis is in logarithmic scale.

CompBio contains four 2.1 GHz Intel Xeon E7-8870v3 processors with 45 MB L3 cache, 1TB of DDR4 RAM, and RAID 1 file systems with rotating disks. All tested software were compiled with GCC v5.3 and OpenMPI v1.10.2 if required.

Distributed-memory experiments were conducted on Iowa State University's CyEnce cluster. Each node contains two 2.0 GHz 8-core Intel Xeon E5-2650 CPUs and 128 GB of RAM. The cluster has quad data rate (QDR) Infiniband interconnect and is supported by a 288TB Lustre file system with 1 MDS and 8 OSTs. All data files are stored on Lustre with 1 MB block size and stripe count of 8. All test binaries were compiled with GCC v4.9.3 and MVAPICH v2.1.7.

For multithreaded programs, we assigned one thread per processor core using `numactl`. For MPI programs (Kmerind and Kmernator), we similarly assigned one MPI process per core via `mpirun`. Assignments are evenly distributed amongst sockets and cluster nodes if applicable. Henceforth, experimental results are discussed using the term "cores".

Unless otherwise specified, the experiments were conducted in canonical mode with DNA 31-mers. For hashed indices, the high and low bits of Google FarmHash outputs are used as upper and lower hash functions, while DHM and DHMM are used as the local hash tables for count and position indices respectively. Each experiment was repeated at least three times, and the fastest time was reported as it most closely reflects system capabilities.

5.1 K-mer Operations

We benchmarked the SIMD accelerated k-mer reverse complement operation using the CompBio system. Fig. 4a summarizes the times to compute reverse complement of one million DNA, DNA5, and IUPAC DNA k-mers for varying k using the x86 SWAR, SSSE 3, and AVX 2 `revcomp` implementations. Fig. 4b summarizes the times for 31-mers.

Overall the SIMD based `revcomp` implementation has a throughput of approximately two microseconds per 31-mer, and scales linearly with the number of machine words in the k-mer data structure. The computation time increases in fixed steps with word count instead of with k directly. SWAR and SSSE 3 implementations were approximately $16\times$ and $21\times$ faster than sequential (SEQ in Fig. 4b). AVX 2 performed comparably to the SSSE 3 for k up to 256 (data not shown) due to the additional overhead incurred when moving bits between 128-bit data lanes.

Based on these observations, we defined an "AUTO" implementation that adaptively chooses the optimal instruction sets at compile time based on k-mer parameters. For small k, the SWAR algorithm is used, while for large k the SSSE 3 implementation is used.

TABLE 4
Time in Seconds to Read a File from the Lustre File System Using 128 Cores, with and without Operating System File Caching

I/O Mechanism	uncached	cached	speed up
MMAP	50.87	29.59	1.72
MPI-IO	57.65	13.26	4.35
POSIX	55.41	2.43	22.80

5.2 Distributed k-mer Parsing

Distributed file reading and k-mer parsing benchmarks were performed on the CyEnce cluster. Three different file access mechanisms were evaluated: MPI-IO, memory mapping (MMAP), and POSIX file access functions. Copies of the same files were used to isolate the effects of file system caching. Parallel k-mer parsing of the $\mathbf{R1}$ dataset scaled nearly linearly for up to $p = 64$, beyond which the network was likely saturated (Fig. 5). MPI-IO and MMAP mechanisms performed similarly given CyEnce's configuration. For 32 and 64 cores, the POSIX mechanism showed an approximately 40 percent advantage.

The time to read and parse the $\mathbf{M4}$ dataset using 128 cores was dramatically improved when the file was previously cached (Table 4). Different I/O mechanisms benefited from caching differently, with POSIX receiving a $22.8\times$ speed up. The long uncached file reading time and the short index construction time (Sections 5.4 and 5.5) suggest that rebuilding a k-mer index from cached sequence data is likely preferable to loading a previously built index.

In subsequent tests, we used POSIX and pre-populated file cache with a "warm up" iteration. File reading times were excluded from the index construction and query times for scalability experiments in Section 5.4, and included for comparisons to existing tools in Section 5.5.

5.3 Effects of Index Parameters

Kmerind provides significant flexibility for users to configure the data structures and algorithms through parameters and compositions. In this section, we briefly examine some of the parameters and their impacts on performance. Fig. 6 summarizes the index operation performance for 4 common parameters. Alphabet, k, and *strand*-mode can be considered as application-driven parameters, while the local map choice is performance-driven.

Among the 3 provided alphabets, DNA provides the best performance as it is compact and allows simple bitwise operations (Fig. 6a) where DNA 5 requires a more complex character shifting algorithm during k-mer parsing from file. The bit length of encoded character is inversely related to the performance of the index, as is the value k. Each short

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

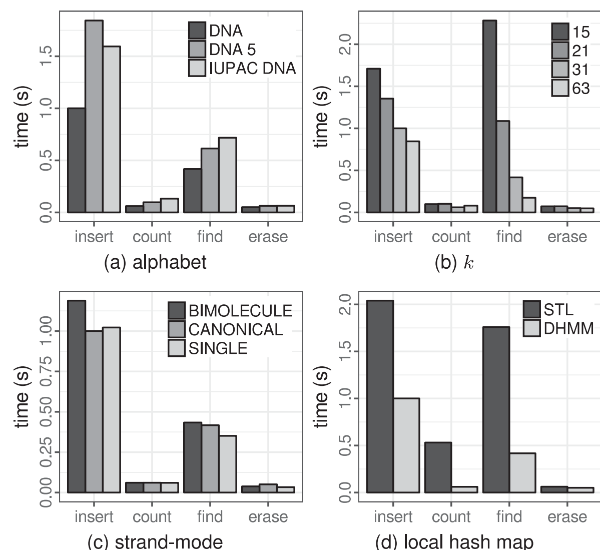


Fig. 6. Parameters affecting the performance of Kmerind's indices. Experiments were conducted using 1,024 cores on CyEnce and a Kmerind canonical hashing position index, configured with Google `farmhash` and DHMM, for DNA 31-mers in the $\mathbf{R1}$ dataset.

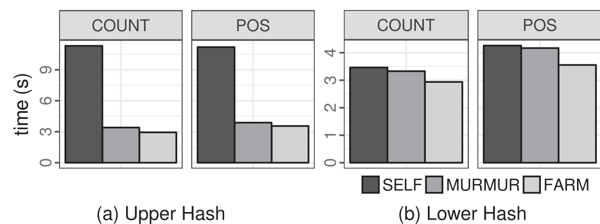


Fig. 7. Hash function impact on performance of DNA 31-mer count and position indices. "SELF" indicates that the k-mer is used as hash value directly. "MURMUR" and "FARM" indicate Murmur3 hash and Google's farm hash, respectively. dataset $\mathbf{M3}$ was processed using 1,024 cores.

read is parsed into $L - k + 1$ k-mers and large k results in fewer k-mers, thus reducing running time (Fig. 6b).

Fig. 6c shows the overhead for the *bimolecule* and *canonical* modes of operation. Bimolecule mode requires canonicalization for the input k-mers as well as the indexed k-mers, thus `revcomp` is applied at least twice for each k-mer during any operation. Canonical mode requires canonicalization once for each input k-mer, while *single-stranded* mode does not require any canonicalization, thus both perform better than the *bimolecule* mode.

For indices that use distributed hash map or multimap, different types of local hash map can be chosen. As shown in Fig. 6d, DHMM consistently outperforms STL's *unordered_multimap* for all except for the `erase` operation. This is because unordered multimap requires the use of linked lists within each bucket.

The choice of hash functions for Kmerind's two-level distributed hash maps can have a strong impact on the performance. We evaluated three hash functions, Google FarmHash (`farmhash`), Murmur Hash 3 (`murmur3`), and k-mer as hash value (`self`), on index insertion at the upper and lower hash levels. In each case, the hash function of one level is varied while the other level is set to use `farmhash`, The experiments were conducted using 1,024 cores and the $\mathbf{M3}$ dataset in single-stranded mode for DNA 31-mers.

Fig. 7a shows that using `self` as upper hash function caused significant performance degradation when compared

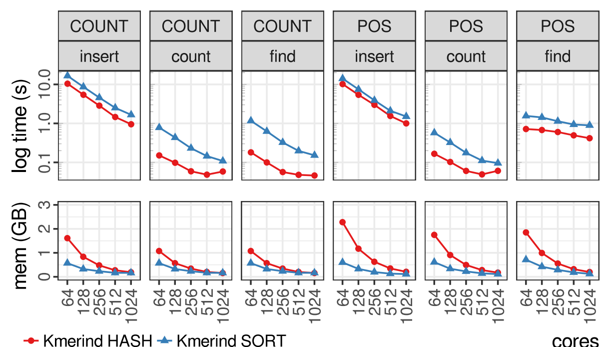


Fig. 8. Strong scaling results for the `insert`, `count`, and `find` operations for the hashing and sorting variants of Kmerind's count and position indices, using canonical DNA 31-mers from dataset $\mathbf{M1}$.

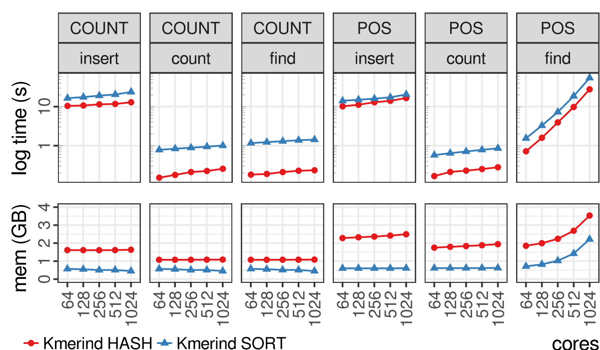


Fig. 9. Weak scaling results for the `insert`, `count`, and `find` operations for the hashing and sorting variants of Kmerind's count and position indices, using canonical DNA 31-mers from datasets $\mathbf{M1}$–$\mathbf{M5}$.

to the results from `farmhash` and `murmur3`. This degradation is attributable to severe load imbalance: while the standard deviations of the number of k-mers assigned to each core were 2,751 and 2,806 for `murmur3` and `farmhash` respectively, for `self` the standard deviation reached 6,662,250. The standard deviations were 41,786, 41,160, and 7,710,595 for `murmur3`, `farmhash`, and `self` based position indices, respectively. As points of reference, the average numbers of k-mers per core were 7,678,665 for counting and 9,007,001 for position indexing.

For the lower level hash function, hash collision and computational overhead are the primary concerns. Fig. 7b shows that `self` and `murmur3` performed similarly for count and position indices, while `farmhash` outperformed both, likely due to more uniform hash value distribution than `self` and better computational efficiency than `murmur3`.

Based on the parameter evaluations, we recommend that, where application allows, a hash map based Kmerind index be used with canonical DNA k-mers. The hash map should be configured with `farmhash` or `murmur3` as the upper level hash function, and DHM or DHMM as appropriate for local storage using `farmhash`. Subsequent scalability and comparison experiments were configured as per these recommendations.

5.4 Scalability

In this section the scalability of each position and count index operation is examined (Figs. 8, and 9). In strong scaling experiments, the total input dataset size $|M|$ is fixed while p is increased to demonstrate an algorithm or software's capability of using additional resources to solve

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

a problem faster. In weak scaling experiments, $|M|/p$ is kept constant, to show the ability of the algorithm and software to solve larger problems by using more resources. Ideal strong scaling means that parallel execution time is $1/p$ times that of sequential execution, while ideal weak scaling translates to constant execution time regardless of p. The `count`, `find` and `erase` operations used 1 percent of the indexed k-mers, sampled randomly, as input. All experiments were performed using datasets **M1**–**M5** on CyEnce.

Kmerind's hashed count and position indices ingested the **M1** dataset in approximately 1 second, and the **M5** dataset in 12.9 and 16.6 seconds respectively using 1,024 cores. Retrieving the counts in the count index required 0.05 and 0.23 seconds for **M1** and **M5**. Retrieving the positions took 0.42 and 28.0 seconds for **M1** and **M5**, respectively.

The sorted array version of the count and position indices ingested the **M1** dataset in 1.66 and 1.5 seconds using 1024 cores, and the **M5** dataset in 23.8 and 20.57 seconds respectively. The position index performed better as count index required an extra integer operation per insertion. Retrieving the counts for the **M1** and **M5** datasets required 0.15 and 1.43 seconds, while retrieving the positions took 0.89 and 54.61 seconds for **M1** and **M5**, respectively.

Overall, the `insert`, `count`, and `erase` operations for both the hashed count and position indices showed similar scaling behavior. Similarly, sorted count and position indices exhibited same scaling trends for these 3 operations. The `find` operations for the corresponding count and position indices showed significantly different scaling behaviors as predicted in Sections 3.3.3 and 3.4.2. The `erase` operation times are not shown in Figs. 8 and 9 as they closely mirror the scaling behavior of the `count` operation.

Figs. 8 and 9 further illustrate the performance characteristics of Kmerind's sorted and hashed indices. Hashed indices consistently and significantly outperformed the sorted indices in time for the index operations, over $6\times$ faster for `find` operations on count indices. At the same time, sorted indices required significantly less memory during execution by nearly a factor of 4 during count and position indices `insert`. This is particularly apparent in the weak scaling experiment results (Fig. 9). The memory advantage of sorted indices decreased with core count for strong scaling as the overheads of local hash maps became less evident (Fig. 8). Note that sorted indices are designed towards a balance between lower memory requirement and acceptable performance and scaling, rather than minimal space requirement.

5.4.1 Analysis of Scaling Behaviors

We examine the `find` operations of the hashed position and count indices in more detail to better understand the scaling behaviors that reflect the algorithmic and complexity differences described in Sections 3.3.2 and 3.3.3.

In strong scaling experiments, `find` for count index reached minima at 512 cores. Fig. 10a shows that the presence of the minima is largely due to communication in the "query" and to a lesser degree to the "resp" steps with complexity $\tau \log(p) + \mu(|M|/p)\log(p)$. For strong scaling, as p increases, the bandwidth term decreases at the rate of $\log(p)/p$, while the latency term increases at a rate of $\log(p)$. For large $p > \mu|M|/\tau$, latency dominates.

Scaling of the `find` operation for k-mer position index is dominated by the "resp" step with complexity $(r|M|/p) + \tau p + \mu(r|M|/p)$ (Section 3.3.3). In contrast to a count index, the "resp" step for the position index has significantly higher latency and computation complexities. In addition, the average k-mer frequency r can increase the bandwidth term contribution for $r > \log(p)$, and highly repeated k-mers can introduce load imbalance amongst cores that further causes the run time to scale sub-optimally.

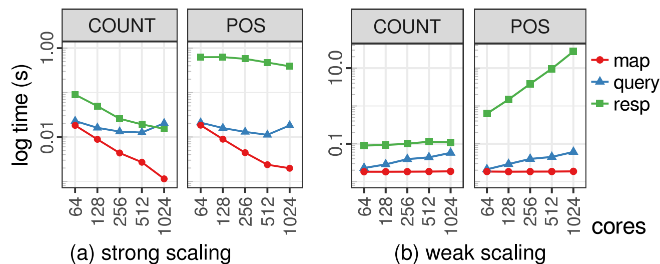


Fig. 10. Strong and weak scaling of internal steps in the `find` operation for hashed count and position indices. The "map", and "query" steps correspond to the `map_to_processor` and `distribute` functions in Algorithms 3 and 5, while "resp" corresponds to all remaining algorithmic steps after the `distribute` step.

Weak scaling experiments showed a slight increase of run time as p increased for the `find` operation (Fig. 10b) for the hashed count index. This is due primarily to the "query" step. As the per-processor data size $|M|/p$ is kept constant in weak scaling experiments, both the latency and bandwidth terms in the communication complexity increase with $\log(p)$.

The `find` operation in a position index scales linearly with p and r according to $\tau p + (\mu + 1)(r|M_i|)$ (Section 3.3.3). Assuming uniform sampling of a true k-mer distribution, r is expected to increase with dataset size, which scales with p for weak scaling experiments. For datasets **M1**–**M5**, r values are 1.11, 1.17, 1.26, 1.37, and 1.54 respectively. Fig. 10b illustrates this linear scaling behavior in the "query" and "resp" steps of the position index `find` operation.

5.5 Comparisons with Existing Tools

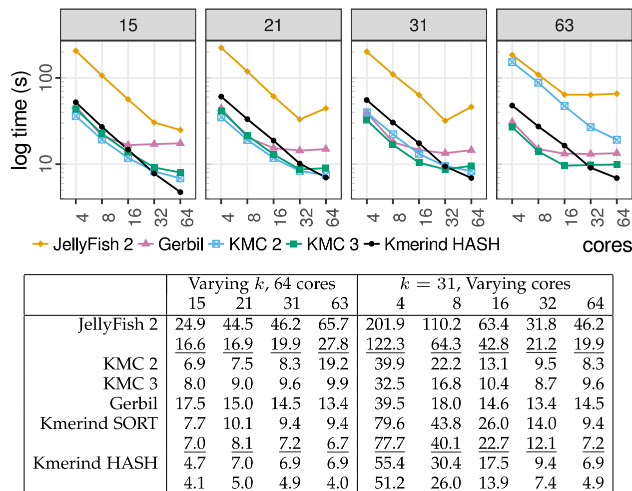
We compared the performance of Kmerind hashed and sorted count indices to existing best-in-class k-mer counting tools on shared- and distributed-memory systems. JellyFish 2 [28] is the *de facto* standard k-mer counter. We also compared to several more recent, state-of-the-art tools including KMC 2 [33], its successor KMC 3 [35], and Gerbil [34]. Kmernator [36] is chosen as it is the only existing, stand-alone, distributed k-mer counter.

5.5.1 Shared-Memory Environment

The CompBio system was used for single-node, multi-threaded testing. Strong scaling experiments were conducted with the 6.3 GB **R1** dataset using 4, 8, 16, 32, and 64 cores for canonical DNA 15-, 21-, 31-, and 63-mers without filtering low frequency k-mers. For Kmerind, we treated CompBio as a distributed-memory system. KMC 2, KMC 3, and Gerbil were allocated 512 GB of main memory and set to memory-only mode where possible to minimize disk usage for intermediate results.

As disk subsystem configurations can vary drastically, we minimized the impact of file I/O by leveraging the operating system cache (Section 5.2), and erasing output immediately after each run due to an observed high overhead to overwrite files on `ext4` file systems. For all software packages, we report the total running time including file input and output. For Kmerind and JellyFish 2, we also report the

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap



	Varying k, 64 cores				$k=31$, Varying cores				
	15	21	31	63	4	8	16	32	64
JellyFish 2	24.9	44.5	46.2	65.7	201.9	110.2	63.4	31.8	46.2
	16.6	16.9	19.9	27.8	122.3	64.3	42.8	21.2	19.9
KMC 2	6.9	7.5	8.3	19.2	39.9	22.2	13.1	9.5	8.3
KMC 3	8.0	9.0	9.6	9.9	32.5	16.8	10.4	8.7	9.6
Gerbil	17.5	15.0	14.5	13.4	39.5	18.0	14.6	13.4	14.5
Kmerind SORT	7.7	10.1	9.4	9.4	79.6	43.8	26.0	14.0	9.4
	7.0	8.1	7.2	6.7	77.7	40.1	22.7	12.1	7.2
Kmerind HASH	4.7	7.0	6.9	6.9	55.4	30.4	17.5	9.4	6.9
	4.1	5.0	4.9	4.0	51.2	26.0	13.9	7.4	4.9

Fig. 11. Strong scaling behavior for counting DNA k-mers in dataset $\mathbf{R1}$. Each plot shows the times in seconds to count a fixed size k-mer (15, 21, 31, 63) on increasing number of cores p (4, 8, 16, 32, 64) in a shared-memory system. The table shows the total and counting-only (underlined) times for either fixed p or fixed k. For readability, Kmerind's sorted count index results are shown only in the table.

counting-only times which excludes result writing. We expect Kmerind's primary application usage pattern to involve constructing and querying in-memory indices, thus the scalability and absolute performance of the counting-only times is of practical importance. Fig. 11 shows the scalability of each software for each k value, while the embedded table shows the running times for fixed $p=64$ and varying k, and fixed $k=31$ and varying p.

Overall, Kmerind's hashed and sorted count indices outperformed the existing tools at high core counts, scaled nearly linearly with p, and behaved well with increasing k. Using 64 cores, Kmerind's hashed count index completed counting 31-mers in $\mathbf{R1}$ in 6.9 seconds and 15-mers in 4.7 seconds, respectively $1.2\times$ and $1.5\times$ faster than the fastest existing k-mer counter for these parameters (KMC 2) and $6.7\times$ and $5.3\times$ faster than JellyFish 2. Kmerind's sorted count index performed competitively against KMC 3, outperformed Gerbil and JellyFish 2, and was bested only by KMC 2. The relative performances at 32 cores was more variable, with Kmerind outperforming Gerbil and JellyFish 2 by 43 and 238 percent respectively, equaling KMC 2, and outperformed by KMC 3 by 8 percent for 31-mers. Kmerind's higher performance at high core count is attributable to our algorithmic design that avoids fine-grained thread synchronizations. On the other hand, the overhead associated with Kmerind's communication and memory operations increases (as $|M|/p$ for strong scaling) with decreasing p, contributing to its lower performance relative to KMC 2, KMC 3, and Gerbil at low core counts.

The counting-only times of Kmerind's hashed and sorted count indices showed near linear scalability for up to 32 cores, beyond which the scalability decreased likely due to MPI communication complexity. The file writing time, as the difference between total and counting-only times, scaled sublinearly with p and remained approximately constant at 2 seconds for 32 and 64 cores. These observations suggest that Kmerind's count index can continue to scale beyond 64 cores, but may be limited by file system performance when result writing is required. In contrast, Gerbil failed to scale beyond 16 cores for all k, while KMC 2, and KMC 3 had

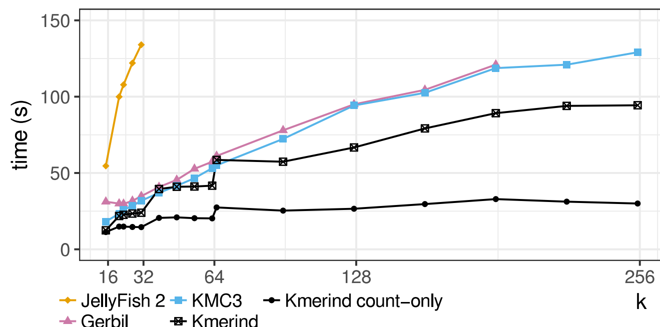


Fig. 12. The time in seconds to count canonical DNA k-mers for increasing k (15, 21, 23, 27, 31, 39, 47, 55, 63, 65, 95, 127, 159, 191, 223, 255). JellyFish 2, KMC 3, Gerbil, and Kmerind's hashed count index were evaluated using dataset $\mathbf{R2}$ on 64 cores. Gerbil failed for $k>191$. JellyFish 2 times for k larger than 31 exceeded 150 seconds and are excluded for readability.

very limited scalability for $k\geq 31$ and $p\geq 16$, indicating that their performance bottlenecks may not be caused by file system limitations. JellyFish 2's counting-only times similarly suggest this hypothesis.

The dependence of KMC 2 on k value was particularly pronounced for $k=63$, where the counting time increased dramatically for all core counts tested. Kmerind, on the other hand, showed relatively constant running time for k values of 21, 31, and 63, and a lower running time for $k=15$. This behavior is attributable to a balance between the widening of k-mer representation from 32 bit to 128 bit, and the reduction in total k-mers in short-read datasets (Section 5.3) with increasing k. For the $\mathbf{R1}$ dataset, the numbers of valid k-mers were 2052-, 1909-, 1670-, and 906-million for k values of 15, 21, 31, and 63, respectively. Gerbil's reduction in running time is likely due to a similar cause, whereas KMC 2 and JellyFish 2's performance degradations suggest inefficiencies in k-mer parsing and comparison operations. We also note that while KMC 3 demonstrated significant improvement over KMC 2 for $k=63$, for low k KMC 2 actually performed better for most p values.

We further evaluated the effects of varying k up to 255 using the $\mathbf{R2}$ dataset on 64 cores, shown in Fig. 12. The values were chosen with consideration of `C++` primitive type sizes. As KMC 3 was reported to significantly improve upon KMC 2's performance for high k values [35], we included KMC 3 only.

Fig. 12 further illustrates Kmerind's low sensitivity to increased in k. Kmerind counted 255-mers in a total of 94.4 second, 30.0 of which was for the *counting-only* time, and was approximately $1.4\times$ faster than KMC 3. At low k, Kmerind's running times increased in a step-wise manner corresponding to the widening of k-mer data structure. For large k, the running time remained relatively constant as the data structure size growth was countered by reductions in k-mer counts. JellyFish 2 was significantly slower than all evaluated software, and failed to count 255-mers. Gerbil performed similarly to KMC 3, but failed for $k>191$.

Table 5 shows the performance of JellyFish 2, KMC 2, KMC 3, Gerbil, and Kmerind's hashed and sorted count indices for datasets of different sizes. All experiments used 64 cores to count canonical DNA 31-mers. The experiments with the metagenomic datasets demonstrated that all tools except for JellyFish 2 scaled nearly linearly with data size. KMC 2 was marginally faster than KMC 3 for all datasets except for $\mathbf{R3}$, while Gerbil's performance lagged behind

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

TABLE 5
Time in Seconds to Count Canonical DNA 31-Mers Using 64 CPU Cores

	Metagenomic			Eukaryotic			Assembled	
	M1	M2	M3	R2	R3	R4	G1	G2
JellyFish 2	133.4	207.5	321.8	127.8	347.3	1465.9	132.6	329.5
KMC 2	22.8	41.9	82.6	29.2	122.1	432.9	30.0	100.3
KMC 3	24.2	45.2	86.5	31.8	99.4	456.0	31.0	102.2
Gerbil	26.3	50.7	97.1	34.8	184.3	696.6	1235.8	153.1
Kmerind SORT	22.3	44.2	86.7	36.3	130.7	515.6	27.7	99.3
	11.0	22.4	45.2	25.6	115.0	477.4	13.9	55.9
Kmerind HASH	16.8	32.4	63.0	24.0	77.9	270.5	21.0	70.6
	6.6	13.2	26.0	14.5	63.1	236.1	9.3	31.4

Underlined values represent the "counting-only" times.

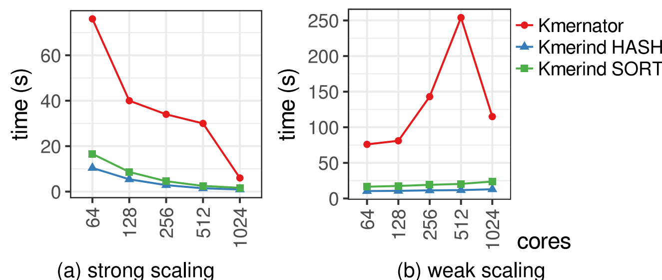
KMC 2 and KMC 3. Gerbil's performance for dataset **G1** was unexpectedly but repeatably poor. Kmerind's sorted index performed comparably to KMC 3 for the metagenomic datasets and the assembled genomes, while the hashed count index outperformed all existing tools for all tested datasets. Kmerind's hashed index counted the **M3** dataset in approximately 63.0 seconds, the **R4** dataset in 270.5 seconds, the assembled human genome (G1) in 21.0 seconds, and the pine genome (G2) in 70.6 seconds. The hashed count index was therefore between $1.3\times$ and $1.6\times$ faster than KMC 2 and KMC 3.

Of the five existing tools tested, only JellyFish 2 includes a command line interface to query the index. KMC 2 and KMC 3 provide an option to find intersection between two indices, but they output the minimum counts for the entries in the intersection. We queried JellyFish 2 and Kmerind hashed count indices using 1 percent of indexed k-mers on a single core, as JellyFish 2 supports single-threaded queries only. JellyFish 2 completed the query in 0.82 seconds while Kmerind was able to do so in 0.11 seconds. The results are not directly comparable as JellyFish 2 requires loading the database file from disk, but they are illustrative of the benefit of Kmerind's in-memory index for on-line queries. This point is further illustrated by Table 5, where the counting-only times of Kmerind's hashed count index is over $3\times$ faster than KMC 2 and KMC 3 for the metagenomic and assembled genome datasets, and approximately $2\times$ for the read sets **R2**, **R3**, and **R4**.

5.5.2 *Distributed-Memory Environment*

While Kmerind performs well in shared memory environments with high core counts, its performance advantages are further extended in a distributed memory environment. We benchmarked index construction for Kmerind's hashed and sorted indices and Kmernator using datasets **M1–M5** and 64 to 1,024 cores on CyEnce.

Fig. 13 shows both of Kmerind's indices were consistently faster than Kmernator by at least a factor of $6\times$ for strong scaling and $8\times$ for weak scaling. Kmerind's hashed count index completed 31-mer counting for the **M1** dataset in 1.0 seconds using 1,024 cores, and Kmerind's hashed index showed approximately linear strong scaling for up to 512 cores, beyond which the parallel efficiency decreased slightly. For weak scaling, Kmerind's hashed index showed a gradual increase of running time as core count increased. In both cases, the behavior is attributable to the $\log(p)$ factor in the collective all-to-all communication complexity.



	time (s)	Core Count 64	128	256	512	1024
(a)	Kmernator	76.0	40.0	34.0	30.0	6.0
	Kmerind SORT	16.6	8.6	4.6	2.5	1.7
	Kmerind HASH	10.4	5.4	2.9	1.5	1.0
(b)	Kmernator	76.0	81.0	143.0	254.0	115.0
	Kmerind SORT	16.6	17.6	19.2	20.5	23.8
	Kmerind HASH	10.4	10.7	11.5	11.8	12.9

Fig. 13. Running times in seconds of Kmerind's hashed and sorted indices and Kmernator for counting canonical DNA 31-mer in strong and weak scaling distributed memory settings. Dataset **M1** was used for strong scaling while sets **M1–M5** were used for weak scaling.

Kmerind's sorted index was also consistently faster than Kmernator, but its scaling behavior was slightly worse when compared to the hashed index, as expected. Kmernator showed a reproducible non-linear scaling behavior for 256 and 512 cores due to unknown cause.

6 Conclusions

K-mer counting and indexing are central to many bioinformatics applications including *de novo* assembly, genome mapping, and error correction. The widespread availability of next generation sequencers and their high throughput and low cost have fundamentally changed the way genomic data are used in biology and medicine. As a consequence, it has become increasingly critical to develop k-mer counting and indexing tools and particularly libraries that can efficiently and scalably operate on very large sequence data.

We present Kmerind, a generic distributed-memory k-mer counting and indexing library that is high performance, flexible, and extensible. To our knowledge, it is the first k-mer indexing library for distributed-memory environments, and the first generic k-mer counting and indexing library. By using distributed memory, entire index can reside in memory for fast access, and additional memory and computational resources can be recruited for larger data sets. Kmerind has also been optimized with efficient SIMD implementation and data structures. We showed that Kmerind indices are capable of index construction and query with linear scaling on distributed systems. On shared memory systems, Kmerind outperforms current best-in-class k-mer counting tools at high core counts and is competitive at moderate core counts.

While the library was implemented using distributed memory parallel algorithms, Kmerind's API has been designed with sequential semantics to facilitate a wider range of application development. The API is C++ templated to allow a user to create custom indices for different applications through composition of predefined logic and user-specified data types. The generic API also allows a developer to extend Kmerind's functionality with novel algorithms and application-specific functional modules and data types in order to improve communication efficiency,

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

minimize local computation and memory utilization, or to integrate novel indexing strategies. For example, a Bloom filter can be used in a pre-processing step to minimize number of k-mers indexed, provided that singleton k-mer exclusion is compatible with the application requirements.

Acknowledgments

This research is supported in part by the National Science Foundation under IIS-1416259, CNS-1229081, CCF-1361053, and the Intel Parallel Computing Center on Big Data in Biosciences and Public Health. *Conflict of interest:* none declared.

References

[1] The Cancer Genome Atlas Research Network, et al., "The cancer genome atlas pan-cancer analysis project," *Nature Genetics*, vol. 45, no. 10, pp. 1113–1120, 2013.
[2] The 1000 Genomes Project Consortium, et al., "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, 2015.
[3] K.-P. Koepfli, B. Paten, S. J. O'Brien, and Genome 10K Community of Scientists, "The genome 10k project: A way forward," *Annu. Rev. Anim. Biosci.*, vol. 3, pp. 57–111, Feb. 2015.
[4] S. Caboche, C. Audebert, and D. Hot, "High-throughput sequencing, a versatileweapon to support genome-based diagnosis in infectious diseases: Applications to clinical bacteriology," *Pathogens*, vol. 3, no. 2, pp. 258–279, 2014.
[5] A. Geskin, et al., "Needs assessment for research use of high-throughput sequencing at a large academic medical center," *PLoS ONE*, vol. 10, no. 6, 2015, Art. no. e0131166.
[6] P. Muir, et al., "The real cost of sequencing: Scaling computation to keep pace with data generation," *Genome Biol.*, vol. 17, 2016, Art. no. 53.
[7] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J. Molecular Biol.*, vol. 215, no. 3, pp. 403–410, 1990.
[8] W. J. Kent, "BLAT the blast-like alignment tool," *Genome Res.*, vol. 12, no. 4, pp. 656–664, 2002.
[9] S. M. Kiełbasa, R. Wan, K. Sato, P. Horton, and M. C. Frith, "Adaptive seeds tame genomic sequence comparison," *Genome Res.*, vol. 21, no. 3, pp. 487–493, 2011.
[10] D. R. Kelley, M. C. Schatz, and S. L. Salzberg, "Quake: Quality-aware detection and correction of sequencing errors," *Genome Biol.*, vol. 11, no. 11, 2010, Art. no. 1.
[11] X. Yang, K. S. Dorman, and S. Aluru, "Reptile: Representative tiling for short read error correction," *Bioinf.*, vol. 26, no. 20, pp. 2526–2533, 2010.
[12] Y. Liu, J. Schröder, and B. Schmidt, "Musket: A multistage K-mer spectrum-based error corrector for illumina sequence data," *Bioinf.*, vol. 29, no. 3, pp. 308–315, 2013.
[13] F. Hach, et al., "mrsFAST: A cache-oblivious algorithm for short-read mapping," *Nature Methods*, vol. 7, no. 8, pp. 576–577, 2010.
[14] Y. Li, J. M. Patel, and A. Terrell, "WHAM: A high-throughput sequence alignment method," *ACM Trans. Database Syst.*, vol. 37, no. 4, 2012, Art. no. 28.
[15] G. Myers, "Efficient local alignment discovery amongst noisy long reads," in *International Workshop on Algorithms in Bioinformatics*. Berlin, Germany: Springer, 2014, pp. 52–67.
[16] S. Batzoglou, et al., "Arachne: A whole-genome shotgun assembler," *Genome Res.*, vol. 12, no. 1, pp. 177–189, 2002.
[17] D. R. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de bruijn graphs," *Genome Res.*, vol. 18, no. 5, pp. 821–829, 2008.
[18] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "ABySS: A parallel assembler for short read sequence data," *Genome Res.*, vol. 19, no. 6, pp. 1117–1123, 2009.
[19] E. Georganas, et al., "HipMer: An extreme-scale de novo genome assembler," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, Art. no. 14.
[20] S. Kurtz, A. Narechania, J. C. Stein, and D. Ware, "A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes," *BMC Genomics*, vol. 9, no. 1, 2008, Art. no. 517.
[21] D. E. Wood and S. L. Salzberg, "Kraken: Ultrafast metagenomic sequence classification using exact alignments," *Genome Biol.*, vol. 15, no. 3, 2014, Art. no. R46.
[22] R. Ounit, S. Wanamaker, T. J. Close, and S. Lonardi, "CLARK: Fast and accurate classification of metagenomic and genomic sequences using discriminative K-mers," *BMC Genomics*, vol. 16, 2015, Art. no. 236.
[23] P. Melsted and J. K. Pritchard, "Efficient counting of K-mers in DNA sequences using a bloom filter," *BMC Bioinf.*, vol. 12, no. 1, 2011, Art. no. 1.
[24] N. Philippe, M. Salson, T. Lecroq, M. Lonard, T. Commes, and E. Rivals, "Querying large read collections in main memory: A versatile data structure," *BMC Bioinf.*, vol. 12, no. 1, 2011, Art. no. 1.
[25] N. Välimäki and E. Rivals, "Scalable and versatile K-mer indexing for high-throughput sequencing data," in *Proc. Int. Symp. Bioinf. Res. Appl.*, 2013, pp. 237–248.
[26] Y. Li and X. Yan, "MSPKmerCounter: A fast and memory efficient approach for k-mer counting," arXiv:1505.06550, 2015.
[27] A.-A. Mamun, S. Pal, and S. Rajasekaran, "KCMBT: A K-mer Counter based on Multiple Burst Trees," *Bioinf.*, vol. 32, no. 18, pp. 2783–2790, Sep. 2016.
[28] G. Marçais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of K-mers," *Bioinf.*, vol. 27, no. 6, pp. 764–770, 2011.
[29] G. Rizk, D. Lavenier, and R. Chikhi, "DSK: K-mer counting with very low memory usage," *Bioinf.*, vol. 29, no. 5, pp. 652–653, 2013.
[30] P. Audano and F. Vannberg, "KAnalyze: A fast versatile pipelined K-mer toolkit," *Bioinf.*, vol. 30, no. 14, pp. 2070–2072, 2014.
[31] Q. Zhang, J. Pell, R. Canino-Koning, A. C. Howe, and C. T. Brown, "These are not the K-mers you are looking for: Efficient online K-mer counting using a probabilistic data structure," *PLoS ONE*, vol. 9, no. 7, 2014, Art. no. e101271.
[32] R. S. Roy, D. Bhattacharya, and A. Schliep, "Turtle: Identifying frequent K-mers with cache-efficient algorithms," *Bioinf.*, vol. 30, no. 14, pp. 1950–1957, 2014.
[33] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, "KMC 2: Fast and resource-frugal K-mer counting," *Bioinf.*, vol. 31, no. 10, pp. 1569–1576, 2015.
[34] M. Erbert, S. Rechner, and M. Mller-Hannemann, "Gerbil: A fast and memory-efficientk-mer counter with GPU-support," *Algorithms Molecular Biol.*, vol. 12, 2017, Art. no. 9.
[35] M. Kokot, M. Dugosz, and S. Deorowicz, "KMC 3: Counting and manipulating K-mer statistics," *Bioinf.*, vol. 33, pp. 2759–2761, 2017.
[36] Kmernator. (2016, Feb.). [Online]. Available: https://github.com/JGI-Bioinformatics/Kmernator
[37] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[38] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *J. Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.
[39] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
[40] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. 41st Annu. Symp. Found. Comput. Sci.*, 2000, pp. 390–398.
[41] P. Flick and S. Aluru, "Parallel distributed memory construction of suffix and longest common prefix arrays," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, Art. no. 16.
[42] Y. Liu, B. Schmidt, and D. L. Maskell, "Parallelized short read assembly of large genomes using de bruijn graphs," *BMC Bioinf.*, vol. 12, no. 1, 2011, Art. no. 354.
[43] Message passing interface (MPI). (2017). [Online]. Available: http://mpi-forum.org/
[44] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A comparison of sorting algorithms for the connection machine CM-2," in *Proc. 3rd Annu. ACM Symp. Parallel Algorithms Archit.*, 1991, pp. 3–16.
[45] P. Flick, "mxx" (2017, Mar.). [Online]. Available: http://patflick.github.io/mxx
[46] R. J. Fisher and H. G. Dietz, "Compiling for SIMD within a register," in *Proc. Int. Workshop Languages Compilers Parallel Comput.*, 1998, pp. 290–305.
[47] Google SparseHash. (2016, Nov.). [Online]. Available: https://github.com/sparsehash/sparsehash

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap

[48] V. Shulaev, D. J. Sargent, and K. M. Folta, "The genome of woodland strawberry (Fragaria vesca)," *Nature Genetics*, vol. 43, no. 2, pp. 109–116, Feb. 2011.

[49] B. Nystedt, N. R. Street, and S. Jansson, "The norway spruce genome sequence and conifer genome evolution," *Nature*, vol. 497, no. 7451, pp. 579–584, 2013.

Tony Pan received the ScB degree in biophysics from Brown University, and the MS degree in computer science from the Rensselaer Polytechnic Institute. Previously, he held positions at General Electric, The Ohio State University, and Emory University. Currently, he is working toward the PhD degree in computational science and engineering in the Georgia Institute of Technology. His research interests include high performance computing, distributed information systems, bioinformatics, and biomedical and imaging informatics.

Patrick Flick received the bachelor's and the master's of science degrees in computer science from the Karlsruhe Institute of Technology in Germany. Currently, he is working toward the PhD degree in computational science and engineering in the Georgia Institute of Technology, Atlanta. His research interests include high performance computing, string algorithms, and graph algorithms.

Chirag Jain received the BTech degree in computer science from the Indian Institute of Technology Delhi. He is currently working toward the PhD degree in the School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta. His research interests include bioinformatics, combinatorial algorithms, and high-performance computing.

Yongchao Liu received the PhD degree in computer engineering from Nanyang Technological University (Singapore), in 2012 and focuses his research on parallel and distributed algorithm design for bioinformatics, heterogeneous computing with accelerators, high performance computing on big data, and parallelized machine learning. He is a research scientist in the School of Computational Science and Engineering, Georgia Institute of Technology. He won two Best Paper Awards at IEEE ASAP 2009 and 2015, respectively, and the PEPI Award from U.S. South Big Data Hub.

Srinivas Aluru is a professor in the School of Computational Science and Engineering, Georgia Institute of Technology. He co-directs the Georgia Tech Interdisciplinary Research Institute in Data Engineering and Science (IDEaS), and co-leads the NSF South Big Data Regional Innovation Hub which serves 16 Southern States in the US and Washington D.C. Earlier, he held faculty positions at Iowa State University, the Indian Institute of Technology Bombay, New Mexico State University, and Syracuse University. Dr. Aluru conducts research in high performance computing, bioinformatics and systems biology, combinatorial scientific computing, and applied algorithms. He is currently serving as the chair of the ACM Special Interest Group on Bioinformatics, Computational Biology and Biomedical Informatics (SIGBIO). He is a recipient of the US NSF Career Award, IBM Faculty Award, Swarnajayanti Fellowship from the Government of India, and the Outstanding Senior Faculty Research Award and the Dean's Award for Faculty Excellence at Georgia Tech. He received the IEEE Computer Society Meritorious Service Award, and is a fellow of the AAAS and the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

orized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 17,2020 at 08:05:04 UTC from IEEE Xplore. Restrictions ap