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

Tony Pan
tpan7@gatech.edu

Patrick Flick
patrick.flick@gatech.edu

Chirag Jain
cjain@gatech.edu

Yongchao Liu
yliu860@gatech.edu

Srinivas Aluru
aluru@cc.gatech.edu

School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA

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, there exist few bioinformatics tools and libraries that can efficiently process the data sets at the current generation rate of 1.8 terabases every 3 days. We present Kmerind, a high performance 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. Using Kmerind, a user can easily instantiate application-specific indices, such as k-mer counter and position index, from biult-in or user-supplied components without extensive high performance computing expertise. Kmerind's k-mer counter performs similarly or better than existing, best-in-class k-mer counting tools even on shared memory systems. In a distributed memory environment, Kmerind counts k-mers in a 120 GB sequence read data set in less than 13 seconds on 1024 Xeon CPU cores, and fully indexes their positions in approximately 17 seconds. Querying for 1% of the k-mers in these indices can be completed in 0.23 seconds and 28 seconds, respectively. To our knowledge, Kmerind is the first k-mer indexing library for distributed memory environments, and the first fully customizable and extensible library for general k-mer indexing and counting. Kmerind is available from `https://github.com/ParBLiSS/kmerind`.

Categories and Subject Descriptors

J.3 [**Life and Medical Sciences**]: Biology and genetics; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*; I.7.2 [**Document and Text Processing**]: Document Preparation—*Index generation*

General Terms

Algorithms, Design, Performance

Keywords

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

BCB'16, October 2–5, 2016, Seattle, WA, USA.

http://dx.doi.org/10.1145/2975167.2975211.

1. INTRODUCTION

Advances in genome sequencing technology has dramatically reduced the cost while improving the throughput of sequencing. A single Illumina's HiSeq X Ten system has the capacity to sequence 18,000 whole human genomes a year at a cost of under $1000 per genome and at a rate of approximately 150 genomes for each 3–day run. As a consequence, biological sequence analysis has become a ubiquitous component of biomedical research and an increasingly important tool for healthcare. The same growth in data size and production rate presents an increasing challenge for timely analysis of biological sequence data.

Central to many bioinformatics tasks is k-mer counting and indexing, which are length k substrings of biological sequences. It is used in data processing tasks including error correction [14, 28, 19], sequence alignment [2, 15], *de novo* assembly [4, 29, 26, 12], and applications that include these as subtasks such as resequencing. Knowledge discovery applications such as repeat detection, sequencing coverage estimation [17] and single nucleotide polymorphism (SNP) identification [14] also require k-mer counting.

Given the wide applicability of k-mer counting and indexing, there exist multiple tools that provide this capability. However, most are standalone tools that target multi-core shared-memory platforms and have performance or memory limitations. In addition, often the tools do no allow online queries, instead store counting or indexing results in files.

We have developed Kmerind, a generic k-mer indexing library, to address both the performance and data scaling challenges. We aim to satisfy the objectives that a Kmerind user should be able to

1. readily scale problem size and/or performance with additional hardware resources,

2. easily configure and extend Kmerind with user specified data types and algorithms, and
3. invoke a consistent set of API for all Kmerind components and functions.

The Kmerind application programming interface (API), algorithms, and implementation directly reflect these objectives. We target distributed memory environments as they allow a user to scale to very large data sizes, up to the total available memory and the aggregate network bandwidth, and to recruit additional computational resources when performance is paramount. Kmerind has been designed to use a bulk-synchronous-parallel model of communication to enforce explicit coarse grain synchronization.

Kmerind classes and functions are templated to allow easy creation of application specific indices. A user can customize k-mer length, alphabet choice, and functions that affect index operations such as the hash function and the k-mer parser. A developer can extend Kmerind's capability by providing novel algorithms and optimized implementations for existing components. All Kmerind indices support the same set of basic operations with sequential semantics and parallel implementations: *insert*, *find*, *count*, and *erase*.

2. RELATED WORK

There has been extensive work related to k-mer counting and indexing, due to its centrality in multiple bioinformatics algorithms. For both counting and indexing, memory is often the primary limitation. Majority of the k-mer counting tools address this issue via external memory techniques, Bloom filters [5], or a combination of both. Most k-mer position indices reduce their memory footprint via enhanced suffix arrays [1] that also supports k-mer counting.

2.1 K-mer Counting

Jellyfish [20], perhaps the most well-known k-mer counting tool, uses a lock-free hash table to support concurrent updates from multiple threads. A bijective hash function allows the lower bits of the key to be reconstructed from the hash bucket id, thus only the upper bits need to be stored. Memory usage is further minimized by widening the data type only when the k-mer frequency is high.

A common approach for reducing the memory footprint is to use disks as external memory. Tools in this group operate with separate partitioning and counting phases. KAnalyze [3] counts k-mers in each file block and stores the intermediate results, which are aggregated during the counting phase. DSK [24], MSPKmerCounter [18], and KMC 2 [7] assign k-mers to on-disk buckets during partitioning, and process each bucket individually during the counting phase.

Another approach to reduce memory consumption is by excluding singleton k-mers, which are assumed to be the results of sequencing errors. BFCounter [21] and Turtle [25] both use Bloom filters to identify previously seen k-mers and count only non-unique k-mers. JellyFish 2 [20] supports this technique optionally. As Bloom filters can introduce false positives, a second scan of the k-mer counts is necessary.

In contrast to BFCounter and Turtle, which counts k-mers exactly, Khmer [30] relies entirely on a generalized probabilistic counting data structure called Count-min Sketch [6]. Count-min Sketch hashes a k-mer to multiple hash tables of varying sizes, and updates the hash table entries with the frequency. When querying, all the hash tables are queried using the k-mer, and the minimum frequency is used. Khmer is memory efficient, but it can over-estimate the frequencies.

Kmerind circumvents single-machine memory limitations by allowing the user to recruit additional nodes in a distributed system. In contrast to disk based tools [3, 24, 18, 7], Kmerind's performance is bound by the faster memory and network rather than disk speed. Whereas techniques involving probabilistic data structures such as Bloom filter [21, 25] or Count-min Hash [30] exclude singleton k-mers or introduce errors, Kmerind supports exact counting and indexing, and therefore is suitable for applications such as error correction. Furthermore, Kmerind provides both counting and indexing capabilities, as well as an extensible library API.

2.2 K-mer Position Indexing

In a suffix tree, a subtree at depth k contains all suffixes sharing the same k-mer as prefix. The subtree's leaves represent positions where a k-mer occurs, and the number of leaves is the k-mer count. Suffix trees, and the equivalent memory efficient enhanced suffix array [1], have been used in Tallymer [17] and Gk-Array [23] for k-mer counting and indexing. FM Index [8] and Compressed Gk-Array [27] further reduce memory consumption via compressed representations of suffix arrays.

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 [11]. However, distributed query processing requires $O(\log(n))$ iterations of sequence comparison, each iteration requiring communication with remote processes. Kmerind instead stores k-mers in hash tables that support local associative look-up of values using k-mers as the 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. To our knowledge, Kmernator [16] is the only distributed memory k-mer counting tool in existence . It is a hybrid MPI-OpenMP application that implements a master-slave model of work assignment at the node and the thread levels. However, Kmernator can only read FASTQ files and count canonical k-mers. A canonical k-mer is defined as the lexicographical minimum of a k-mer and its reverse complement.

Distributed memory assemblers often embed specialized k-mer indexing and counting capability for removing the erroneous k-mers and constructing the de Bruijn graph. ABySS [26] is one of the first distributed memory assemblers. It constructs a distributed de Bruijn graph by first distributing k-mers among MPI processes, and then identifying the edges by querying for the 8 possible neighbors of each k-mer. ABySS then traverses the graph by following each k-mer's edges. These operations can be viewed as k-mer index insert and query operations. HipMer [12] similarly constructs a distributed de Bruijn graph, and improves upon ABySS by discarding the erroneous k-mers using a distributed Bloom filter. Neighbor k-mer queries are not needed as HipMer builds the graph edges directly using $(k+2)$-mers from the input sequences. Graph traversal requires neighbor queries and HipMer uses UPC one-sided communication to increase computation and communication overlap. In both assemblers, the k-mer indexing capabilities are specialized for assembly only, and neither provides reusable indexing APIs.

3. PARALLEL K-MER INDEXING

We designed and implemented Kmerind as a distributed memory library based on the objectives listed in Section 1. Kmerind's algorithms are designed to be efficient in both computation and communication complexities. It leverages bulk-synchronous parallel communication primitives with explicit synchronization semantics to avoid contention, fast hash tables to store k-mers and associated data, and vectorized operations on k-mers. Kmerind does not use multithreading since thread-safety would incur additional overheads for this highly data parallel task. An index is kept completely in memory for the duration of its use.

The following notations are used in subsequent discussions. The number of processors or cores used is denoted as p. Communication complexity is described in terms of latency τ, bandwidth $1/\mu$, and message size m. The complexity of point-to-point communication is $O(\tau + \mu m)$. The number of times a k-mer appears in a data set is referred to as its *count* or *frequency*. We use the term *distinct* to describe k-mers with different character sequences, and *unique* to refer to k-mers with counts of 1. The sets of all k-mers and distinct k-mers indexed by process i are referred to as N_i and U_i, and the input k-mers for an index operation on processor i is denoted as M_i. The un-subscripted N, U, and M denote the corresponding complete k-mer sets across all processors. The size of a set is represented by $|\cdot|$. We note that while N and M are concatenations of the sets N_i and M_i across all processors, the same is not necessarily true for U, as a locally distinct k-mer may appear on multiple processors. The average global and local k-mer frequencies are denoted by $r = |N|/|U|$ and $r_i = |N_i|/|U_i|$.

In section 3.1, we describe the distributed file parsing and k-mer generation algorithm. We then describe in Section 3.2 Kmerind's distributed data structures and the associated k-mer mapping and data distribution functions. As counting k-mers and indexing their positions are two common tasks, in Sections 3.3 and 3.4, we describe the parallel distributed algorithms for the operations of k-mer count and position indices. While the choice of k is dictated by the application, its effects on index performance is discussed as well.

3.1 Distributed K-mer Parsing

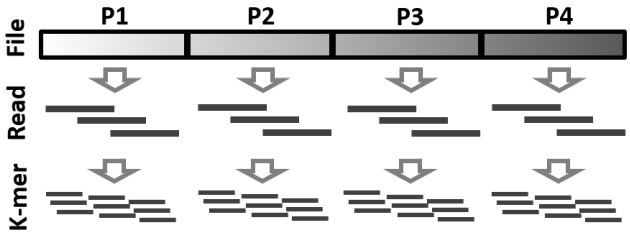


Figure 1: Parallel FASTA and FASTQ file reading in distributed environment.

Kmerind supports parallel reading of FASTQ and FASTA files. The FASTQ format is used primarily for storing sequencer output and contains numerous short sequences called *reads*. The FASTA format can represent short sequences or partial or complete genomes. We denote the length of the file as F and the average sequence record length, including header and other data, as R, and the average length of the sequence data in the records as L.

Parallel file reading and k-mer parsing proceeds in 3 steps: *file partitioning*, *sequence segmentation*, and *k-mer generation* (Figure 1). In the *file partitioning* step, the file is divided into approximately equal partitions of F/p bytes and loaded in parallel into memory. Subsequent file data processing occurs in memory only. Kmerind partitions the input file in order to process files that may cause the memory limits of a single machine to be exceeded. Parallel file loading can benefit from the use of local solid state drives (SSDs) due to their high read performance. In a distributed memory environment with a parallel file system, however, the benefit of SSD is less clear as the network bandwidth becomes the limiting factor for large p.

Kmerind currently supports the common FASTQ format where a sequence record consists of 4 lines: sequence header, sequence data, quality score header, and quality scores. It requires a sequence record to be shorter than the file partition size so it does not span more than 2 partitions. A sequence may still be split between 2 processors, however. To simplify subsequent processing, we realign the partitions to sequence boundaries. Each processor performs linear scans of average length R from the beginning and the end of its partition to find the starting offsets of the first complete sequences. The local partition is adjusted to include the file data between the 2 offsets. The file data at the end of the partition is read incrementally during this search to minimize disk access and memory usage.

For the FASTA format, sequence lengths and counts can vary widely between files, and a single sequence may span multiple processors. We instead maintain exact block partitioning of size F/p with (k-1)-byte overlap to avoid load imbalance. Each partition is scanned linearly and the starting offsets of all sequence records are stored locally. The offsets of the split sequences are replicated to all the involved processors via forward and reverse parallel prefix scans using $max(\cdot)$ and $min(\cdot)$ as operators, respectively. Global sequence ids are computed via parallel prefix sum. Each processor communicates one element per parallel prefix scan, whose time complexity $O(\tau log(p) + \mu log(p))$ is small relative to overall k-mer parsing time.

In the *sequence segmentation* step, the in-memory partitions are segmented into sequence objects, each containing an id and the sequence's starting and ending offsets in the file. For a FASTQ file, a linear scan for 4 complete lines is sufficient to obtain a sequence object. For a FASTA file, each sequence object is retrieved in constant time from the previously computed sequence id and offset arrays.

Finally, during the *k-mer generation* step, k-mers are extracted in linear time from a sequence object in a sliding window fashion. User supplied logic may simultaneously generate other data such as positions and quality scores.

Table 1 summarizes the complexity of each of the steps of parallel file reading and k-mer generation for FASTQ and FASTA files. The overall time complexity for parallel file reading and k-mer generation is $O(F/p)$.

It is important to note that a user's choice of k affects the size of an index and the average frequency. Each sequence record is parsed into $L - k + 1$ k-mers as the k-mer does not extend past sequence data boundaries, therefore a file produces $|N| = (F/R)(L - k + 1)$ k-mers. The maximum number of distinct k-mers, $|U| = \min(4^k, (F/R)(L-k+1))$,

Table 1: Time complexities for parallel FASTQ and FASTA file reading and k-mer generation.

	FASTQ	FASTA
partition file	$O(F/p + R)$	$O(F/p + k)$
segment sequences	$O(F/p)$	$O((F/p + k)/R)$
generate k-mers	$O(F/p)$	$O(F/p + k)$

is bounded by the organism's genome size, the number of k-mers in the file, and the size of the k-mer space. For the human genome with 3 billion bases, the k-mer space size exceeds the genome size at $k \geq 16$. Below $k = 16$, $|U|$ is an exponential function of k, and the average k-mer frequency r increases as k decreases. Above $k = 16$, r decreases while both $|N|$ and $|U|$ decrease linearly with k.

3.2 Distributed Hash Table

All Kmerind indices are implemented as lightweight wrappers to 2-level distributed hash tables with k-mer as the key and count, position, or other user-specified data as the value. Upper level hashing maps the k-mers to the processors, and the lower level consists of a local hash table for storing k-mers and associated data. The use of 2-level distributed hash table provides flexibility as different hash functions can be chosen for the two levels, and different local containers to be used. It also allows a constructed k-mer index to easily be redistributed to a different set of processors, potentially of different size, without first writing the index to disk.

We chose hash table for local data storage due to its expected $O(1)$ access time complexity. In the upper level, we partition the entire k-mer space via surjective hashing where a k-mer is uniquely and deterministically assigned to a processor, which avoids broadcasting during insertions and queries. Well chosen hash functions at both levels are critical and should satisfy two criteria. First, a hash function should produce uniformly distributed hash values to avoid load imbalance at the upper level and minimize collisions in the lower level. Second, the upper and lower levels hash functions should produce uncorrelated hash values. For example, if the same hash function were used at both levels, a processor with id $i \equiv hash(\cdot)\%p$ would receive k-mers with identical least significant $\lceil \log(id) \rceil$ bits in its hash values. The inserted k-mers would then cluster in buckets with id $j \in \{j\%p \equiv i\}$ in the local hash tables.

Kmerind provides *insert*, *count*, *find*, and *erase* operations for counting and position indices. Kmerind index operations process arrays of input collectively in order to minimize communications overhead and facilitate optimization. Each operation requires the input k-mers to be assigned and then sent to the remote processors, and any results to be sent back to the originating processors. We define two function interfaces, *map_to_process* and *distribute*, to encapsulate these two commonly used tasks.

The *map_to_process* task assigns k-mers to processors using the upper level hash function, and rearranges input k-mers into buckets in order to simplify communication. The number of k-mers for each processor is counted via a linear scan, and the bucket offsets are computed using an exclusive prefix scan. The k-mers and associated values are then copied into contiguous regions in the output array at the bucket offsets during a second linear scan. This is a local operation with time complexity that is linear in the local input size and processor count, i.e., $O(|N_i| + p)$.

The *distribute* function sends k-mers and associated data to target processors using a single collective all-to-all communication call. The time complexity of *distribute* depends on the implementation of all-to-all communication. Here we assume all-to-all communication uses hypercubic permutations with complexity of $O(\tau \log(p) + \mu m \log(p))$.

An important consideration in parallel k-mer indexing is load balancing between processors. Imbalance primarily arise from two sources. If the distinct k-mer set U is not uniformly distributed in the k-mer space, the k-mers may be mapped non-uniformly to the processors. Well chosen hash functions can help to alleviate this concern. A second source of imbalance is the non-uniform distribution of k-mer sets M and N, even when U is uniformly distributed. This is the consequence of non-uniform k-mer frequencies and results in computational and communication imbalance during queries. We describe this situation and outline an approach to address it in Section 3.4.

As described in Section 3.1, the choice of k affects the size of the complete and distinct k-mer sets, as well as the average k-mer frequency. The value of k consequently affects the space and time complexities of the distributed hash table and its operations indirectly through $|N|$, $|U|$, and r. High average k-mer frequencies increase collisions at both levels of the distributed hash table, and in the case of position index, the number of associated values for a k-mer. A high $|U|$ value increases the number of local hash table entries required. However, as k is chosen to satisfy application requirements rather than as tuning parameter for our parallel algorithms, the complexities discussions in subsequent sections are in terms of N, U, M, r, and related quantities.

3.3 Distributed K-mer Count Index

Kmerind's count index is a light weight wrapper around a distributed hash table (Section 3.2) with $\langle k$-mer, count$\rangle$ as value type, where the k-mer $\in U$, and *count* stores the number of occurrences of the associated k-mer. For k-mer counting, the distributed hash table is modeled as a reduction map with + operator over the *count* field. In this Section, we present the algorithms for the four distributed hash table operations in the context of k-mer counting.

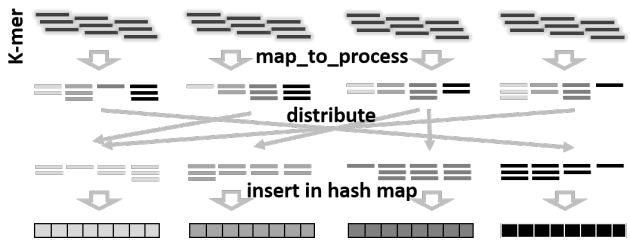


Figure 2: Inserting k-mers into the Kmerind's distributed index.

All operations of the hash table follow the same basic flow. Index insertion is illustrated in Figure 2. The *erase* operation follows a similar flow, while the *count* and *find* operations includes a final communication step to send responses. The Master Algorithm in Algorithm 1 outlines the complete distributed hash table query and modification pipeline and step-wise complexities. The Master Algorithm distributes the input to target processors using *map_to_process* (Line 2) and *distribute* (Line 3) functions described in Section 3.2.

Target processors then process the received k-mers and data locally (Line 6). Any query results (Line 7) are communicated back to the source processes (Line 9), if required, using the inverse of *map_to_process* mapping. Duplicates in input may be removed as needed for the distributed hash table operation before *map_to_process* to reduce subsequent communication volume (Line 1), or after *distribute* to reduce computational load on the local hash table (Line 4).

Algorithm 1: Master Algorithm for distributed hash table operations. Complexities of steps inside a loop are colored gray, while loop total is shown on the first line.

Input: M_i: input array of k-mers
Input: ht: local hash table
Parameter: h: upper level k-mer hash function
Parameter: i: current processor rank
Output: R_i: output array of $\langle k$-mer, value$\rangle$ pairs

1	$Q_i \leftarrow$ distinct(M_i)	$O(\lvert M_i\rvert)$
2	$Q_i \leftarrow$ map_to_process(Q_i, h)	$O(\lvert Q_i\rvert)$
3	$D_i \leftarrow$ distribute(Q_i)	$O(\tau \log(p) + \mu \lvert Q_i\rvert \log(p))$
4	$D'_i \leftarrow$ distinct(D_i)	$O(\lvert D_i\rvert)$
	`// process query k-mers`	
5	**for** $x \in D'_i$ **do**	$O(\lvert D'_i\rvert)$
6	$\quad v \leftarrow ht.operation()$	$O(1)$
7	$\quad R'_i$.append$(\langle x, v\rangle)$	$O(1)$
8	**end**	
9	$R_i \leftarrow$ distribute(R'_i)	$O(\tau \log(p) + \mu \lvert D'_i\rvert \log(p))$

Table 2: Customization of the Master Algorithm for each count index operation. An index operation requires a line in the Master Algorithm if the corresponding table cell is marked with •.

	index operation			
line	*insert*	*count*	*find*	*erase*
1		•	•	
4				•
6	$++ht[x]$	ht.count(x)	ht.find(x)	ht.erase(x)
7		•	•	
9		•	•	

The reduction map *insert*, *count*, find, and *erase* operations customize the steps in the Master Algorithm according to Table 2. For *insert*, per-process duplicate removal (Line 1) provides limited benefit as the expected number of duplicated k-mer on a processor, $(1/|U|)(|N|/p) = r/p$, decreases with increasing p. For a typical whole genome sequencing data set with 30× coverage and $p = 32$, we expect that most k-mers are locally distinct. Experiments in Section 5 therefore do not invoke Line 1 during *insert*. In addition, Line 4 is functionally redundant as local hash table insertion performs the same logic and Lines 7 and 9 are not required.

For *count* and *find*, the second *distinct* is bypassed so that a response is generated for every query k-mer from each source processor. The response *distribute* step (line 9) has the same complexity as the query *distribute* step since one response element is returned per query k-mer for counting. As *count* operation returns the number of $\langle k$-mer, count$\rangle$ pairs with matching k-mer value, while the *find* operation returns the value of the *count* field, k-mer counting relies on the *find* operation. *Erase* operation is similar to *insert* except that only globally distinct input k-mers are processed (Line 4), so that each distinct k-mer is erased only once.

We utilize hash tables with appropriately selected hash functions and dynamic arrays with amortized $O(1)$ insertion time to ensure that local operations have expected $O(1)$ time complexity per k-mer. The *distinct* function internally uses a temporary hash table for the same reason. We enforce that input data for index operations are block partitioned, therefore $|M_i| = |M|/p$, and $|M_i| \geq |Q_i|$. Each processor receives a k-mer set D_i after the *distribute* step. As the subset D'_i contains distinct k-mers, the local k-mer frequency after distribution is $r_i = |D_i|/|D'_i|$ for processor i. The Master Algorithm therefore has an overall time complexity of $O(|M|/p+|D_i|+\tau \log(p)+\mu(|M|/p+|D'_i|) \log(p))$ for count index operations, with the first two terms corresponding to computation, and the last two representing collective all-to-all communication.

Load balance is ensured by uniform distribution of D_i and D'_i. If the condition that k-mers in M are distributed uniformly to the processors is met, then the global *distinct* call is computationally balanced with $|D_i| = |M|/p$. If the second condition that M has the same distribution as U is met, then the number of distinct k-mers received for local processing is approximately the same for all processors $|D'_i| = O(|M|/p)$. The second condition also implies that local average k-mer frequencies after *distribute* are identical between processors, $r_i = |M|/|U|$. When both conditions are met, the overall time complexity simplifies to $O(|M|/p+\tau \log(p)+\mu(|M|/p) \log(p))$. A well chosen upper-level hash function that maps k-mer to hash values uniformly can help to ensure that these conditions are met.

3.4 Distributed K-mer Position Index

Similar to count index, Kmerind's position index is a light weight wrapper around a distributed hash table with $\langle k$-mer, position$\rangle$ as value type. Unlike count index, the tuples do not need to have distinct k-mers and a multimap is used for local storage. The *insert*, *count*, and *erase* operations follow the same Master Algorithm (Algorithm 1) and operation-specific customizations (Table 2) as described in Section 2.1. The complexities of the local hash table operations (Line 6) depend on the multimap implementation, however. We use a customized version of Google SparseHash Dense Hash Map with constant expected time per k-mer *insert*, *count*, and *erase* operations. Its *find* operation scales with the k-mer frequency, as described in Section 4.2. The run time complexities of the *insert*, *erase*, and *count* operations for position index are identical to those for the count index, $O(|M|/p+|D_i|+\tau \log(p)+\mu(|M|/p+|D'_i|) \log(p))$ in general and $O(|M|/p + \tau \log(p) + \mu(|M|/p) \log(p))$ when the k-mer uniform distribution conditions are met.

In contrast to the *find* operations for the count index, a position index's *find* operation returns each tuple instance of a matching k-mer to the querying processor, thus amplifying the computational, memory, and communication requirement on processor i by the average frequency of the k-mers, r_i, in the indexed k-mer set N. Direct application of Algorithm 1 results in complexity $O(|M|/p+|D_i|+\tau \log(p)+\mu(|M|/p+r_i|D'_i|) \log(p)+r_i|D'_i|)$ where the query processing and result communication time and space complexities are increased by a factor of r_i.

Furthermore, if we assume that the uniform input k-mer distribution condition is met, then a highly repeated query

k-mer x, with frequency of s_x in the query set M and r_x in the indexed data N, is expected to be sent from $min(p, s_x)$ processors. After the *distribute* step, the target processor holds all $min(p, s_x)$ instances of x, each from a different processor. Query processing a single query k-mer x on the target processor requires $O(s_x r_x)$ time and similar space for its results, and $O(\tau \log(p) + \mu s_x r_x \log(p))$ time for communication. The complexity therefore has second order dependence on the repeat rate distributions of the query k-mer set M and the indexed k-mer set N. If M and N have identical repeat rate distributions, then the complexity is quadratic in the repeat rate of each k-mer, r_x^2. Non-uniformity in repeat rate distribution can quickly cause load imbalance in computation, memory usage, and communication for Algorithm 1, where collective all-to-all communication is sub-optimal.

Algorithm 2: *Find* all ⟨k-mer, value⟩ pairs with matching k-mer in Position Index

Input: M_i: array of query k-mers
Input: ht: local hash table
Parameter: h: k-mer hash function
Parameter: i: current processor rank
Parameter: r_i: average processor i k-mer frequency
Output: R_i: output array of k-mers and data

```
1  Q_i ← distinct(M_i)                                   O(|M_i|)
2  Q_i ← map_to_process(Q_i, h)                          O(|Q_i|)
3  D_i ← distribute(Q_i)                  O(τ log(p) + μ|Q_i| log(p))
   // count results per source process
4  C_i ← empty array of size p
5  for j = 0 to (p − 1) do                               O(|D_i|)
6  |  D_ij ← subset of D_i from processor j
7  |  for x ∈ D_ij do                                    O(|D_ij|)
8  |  |  C[j] += ht.count(x)                            O(1)
9  |  end
10 end
11 C_i ← distribute(C_i)                      O(τ log(p) + μ log(p))
12 c ← sum(C_i)                                          O(p)
   // find k-mers in index per source process
13 R_i ← empty array of size c
14 for j = 0 to (p − 1) do                   O(τp + (μ + 1)r_i|D_i|)
15 |  k ← (i + j) mod p
16 |  D_ik ← subset of D_i from processor k
17 |  T ← empty array
18 |  for x ∈ D_ik do                                    O(r_i|D_ik|)
19 |  |  T.append( ht.find(x) )                          O(r_i)
20 |  end
21 |
22 |  T ← send(T, t)                                     O(τ + μr_i|D_ik|)
23 |  R_i.append(T)                                      O(r_i|D_ik|)
24 end
```

Instead, Kmerind's position index's *find* operation uses a modified algorithm (Algorithm 2) that processes query k-mers from one source processor at a time (Line 4) and send results immediately (Line 22) before proceeding to the queries from the next processor. This approach avoids the quadratic query result space requirement for highly repeated k-mers, reduces the query processing time and space complexity from $O(r_i|D_i|)$ to $O(r_i|D_i|/p)$ and the communication message sizes similarly. Instead of collective all-to-all communication, non-blocking point-to-point communication is used, which also allows communication to overlap query processing computations.

The modified *find* algorithm for the position index has complexity that is dominated by query processing and results communication. The overall complexity is $O(|M|/p + r_i|D_i| + \tau(p + \log(p)) + \mu((|M|/p)\log(p) + r_i|D_i|))$. Under uniform k-mer distribution ($|D_i| = |M|/p$) and uniform local average k-mer frequency ($r_i = r$) conditions, the overall time complexity simplifies to $O(r|M|/p + \tau p + \mu(\log(p) + r)(|M|/p))$.

4. SOFTWARE ARCHITECTURE

The Kmerind library has been designed with a tiered architecture (Figure 3), built on C++ 11 language features, Standard Template Library (STL) containers and algorithms, and MPI functions. Each tier defines templated functions and class interfaces as well as generic implementations. The templated API is designed to allow functionality by composition and extension by specialization and inheritance. Where possible, the API presents sequential semantics for simplicity, and encapsulates distributed memory algorithm details.

At the lowest level, *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 as interchangeable modules for reading files of different formats and generating ⟨k-mer, value⟩ pairs of different types.

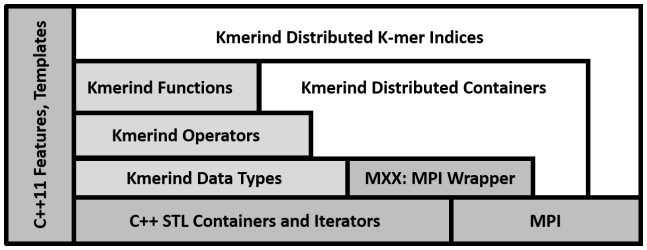


Figure 3: Kmerind Library's tiered architecture.

The *Distributed k-mer Index* layer contains k-mer indices, which are implemented as light-weight wrappers for *Distributed Containers*. Kmerind provides distributed hash map, multimap, as well as map that performs reduction on insertion, and its counting map specialization. Different local hash tables, such as DHMM described in Section 4.2, can be used with the distributed maps.

A user chooses library-provided or custom-developed components as template parameters during index instantiation to define its behavior. Through template specialization, function overloading, or class inheritance, custom implementations of generic library API can be created. Kmerind's flexibility and extensibility is due largely to this approach. For example, a simple de Bruijn graph can be constructed by defining the value type to be edge labels and extending the base k-mer parser to generate the edges.

4.1 K-mer Representation

In Kmerind k-mers are specified via length k and alphabet Σ. Kmerind does not place restrictions on values of k, including whether k is odd. Kmerind currently define 3

primary alphabets, DNA, DNA5, their RNA variants, and IUPAC DNA. A user can also supply custom alphabets. Each alphabet defines the allowable ASCII characters for representing nucleotides. For DNA, the character set 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. An alphabet also defines the complement mapping for each nucleotide.

Kmerind uses a compressed k-mer representation. Each nucleotide is encoded using the minimum 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 as a bit vector of length kb in an array of machine words, with unused bits in the most significant positions. Operations on k-mers has complexities that depend linearly on k and the machine word size, and is vectorized as described in the remainder of this section.

As DNA is double stranded, given a k-mer x, the corresponding k-mer on the opposite strand $\bar{x}$ can readily be obtained via the reverse complement operation, *revcomp*. A canonical k-mer, $\tilde{x}$, is the lexicographical minimum of x and $\bar{x}$. Kmerind does not explicitly model k-mers as double stranded, instead accounting for the double stranded nature in indexing operations. Kmerind can store and query the input k-mers as is (single strand mode), convert all k-mers to canonical (canonical mode), or store input k-mers as is but treat x and $\bar{x}$ as equivalent for queries (bimolecule mode). Note that bimolecule mode is handled completely on-demand by an index's hash and comparison functions. As *revcomp* is a central operation, it has been accelerated using Single Instruction Multiple Data (SIMD) instructions and the SIMD Within A Register (SWAR) [9] pattern where only x86 instructions are used.

Revcomp 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. Words in a k-mer are processed in linear order. The *pshufb* instruction from SSSE 3 and AVX 2 is used for byte- and character-reversal. It copies bytes from a register into the output in $O(1)$ time using a second register as lookup index. The x86 instruction set lacks a *pshufb* equivalent, so for character reversal, we swap the upper and lower bit blocks using the *mask-shift-or* pattern, halving the block size in each of the $\log(|word|/b)$ iterations. *Bswap* is used for x86 byte reversal. Table 3 summarizes the instructions used.

Table 3: SIMD and x86 instructions used during k-mer reverse complement. *Word* refers to the input data. *Index* is a constant containing the ranks in reversed order. *LUT* is an alphabet-specific lookup table containing the character-by-character reversed bit patterns of the look up table index, e.g. 01 00 for position 1 (00 01) in DNA LUT.

PHASE	x86	SSSE 3, AVX 2
reverse bytes	bswap(word)	pshufb(word, index)
reverse bits	mask-shift-or	pshufb(LUT, word)

To accelerate character complement, we designed SIMD friendly bit-encoding schemes for the Kmerind alphabets. Specifically, the encodings are chosen such that the complement of a character can be computed via simple vectorized functions. For DNA5 and IUPAC DNA, the *bit reverse* function is chosen, while for DNA alphabet, *bitwise negation* is used (Table 4). For example, for DNA 5, 'C' is mapped to 011, whose bit-wise reversal is 110, corresponding to 'G' in the alphabet. For encodings where complements are computable via bit-reversal, the character reversal mechanism can be extended to compute reverse complement in one step. This approach also 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.

Table 4: SIMD-friendly bit encoding for DNA, DNA5, and IUPAC DNA characters and corresponding character complement method. For IUPAC DNA alphabet, not all characters are shown. and the functions for computing the complements.

	Character		Complement		Complement
	Char	Bits	Char	Bits	Method
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	

4.2 Local Hash Table

Kmerind incorporates Google SparseHash's Dense Hash Map (referred to as *DHM*) [13] 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 5 summarizes the decision tree for selecting the strategy to generate the *empty* key for *DHM*. *Deleted* key selection is similar.

Table 5: Strategies for choosing k-mers as the *empty* key for *DHM*. 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.

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

Three conditions are evaluated based on k-mer parameters. First, if the alphabet chosen is DNA5, bit patterns that are not mapped to alphabet characters, (010, 101), are used to create *deleted* and *empty* keys. If the alphabet is not DNA5, and there are unused bits in a k-mer's data structure, then the unused bits indicate *deleted* and *empty* keys.

If all bits are used, we choose k-mers that are never inserted or queried in the index. For canonical mode indices,

the lexicographically smaller of a k-mer and its reverse complement is inserted into the index, implying that any lexicographically larger k-mer can be used as the *empty* or *deleted* key. We choose the lexicographically largest k-mer, $TTT \ldots TTT$ as the *empty* key.

Finally, if none of the three previous conditions are met, then any k-mer may be inserted into the index. In this case, we partition the k-mer space into upper and lower halves based on the most significant bit in the k-mer, and use 2 separate DHMs, one for each half. The lexicographically largest and smallest possible k-mers are chosen as keys for the lower and upper DHMs, respectively. For all cases, The same keys are used for all DHMs in a distributed index.

We also extended *DHM* into a multimap, which is required by k-mer position index. *Dense Hash MultiMap*, or *DHMM*, allows multiple values per k-mer key through an indirection to secondary arrays. *DHMM* stores singleton k-mers in one array, referred to as *SA*, and duplicated 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* associates k-mers with *SA* or *MA* positions, differentiated by the sign bit of the position value. 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 duplicated k-mers minimizes the number of memory allocations for inner arrays of *MA*.

DHM has amortized $O(1)$ insertion time complexity as it can dynamically resize as needed. In practice, Kmerind preallocates the local hash table if possible. *DHM*'s find, count and erase operations have expected $O(1)$ complexity. Insertion in *DHMM* has an amortized $O(1)$ time complexity as the *SA* and *MA* array as well as the internal *DHM* may resize as needed. Counting requires constant time as the count for singleton k-mers is 1, while the counts for repeated k-mers can be retrieved from the corresponding inner arrays directly. 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 average size of the output, $O(r)$.

5. EXPERIMENTAL RESULTS

We examined the performance of Kmerind and those of a select subset of existing tools. Kmerind has been implemented as a header-only C++ 11 library. MPI 2 collective and point-to-point operations were used for inter-node communication and the *mxx* library [10] provided convenience wrappers for MPI functions and large message support (greater than 2 billion elements). Kmerind is available from `https://github.com/ParBLiSS/kmerind`.

Data sets used for the experiments include a filtered human genome read file (1000 Genome Project HG00096), the Iowa Cornfield Soil Metagenomic data set (Joint Genome Institute 402461), the complete human genome (1000 Genome Project reference genome version GRCh37), and the *Picea abies* pine genome (Nystedt et al [22]). The HG00096 read data set is a 6.3 GB FASTQ file and identified as **R1**. The humane and pine genomes are FASTA files with sizes 2.9 GB and 12.4 GB, and are identified as **G1** and **G2**, respectively. The original metagenomic file is approximately 440 GB in size. We extracted 7.5, 15, 30, 60, and 120 GB regions via truncation. These are referred to as **M1**, **M2**, **M3**, **M4**, and **M5** data sets, respectively.

Unless otherwise stated, the tests were run with DNA 31-mer. Kmerind indices were configured in canonical mode and used the high and low bits of Google farmhash hash values for the upper and lower levels of the distributed hash table. DHM and DHMM are used as the local hash table for count and position indices, respectively, as they achieved up to 4x speedup in testing over STL's *unordered_multimap* for position index count and find operations. All figures are rendered with $\log(corecount)$ on the x-axis, and $\log(time)$ on the y-axis in seconds, unless specially noted.

Both strong and weak scaling experiments were conducted. In strong scaling experiments, the total input data set size $|M|$ is fixed while p is increased to demonstrate the software's capability to use additional resources to solve a problem faster. In weak scaling experiments, p is increased with $|M|$, such that $|M|/p$ is constant, to show the ability of the software to solve larger problems by using more resources. Ideal strong scaling means that parallel execution time is $1/p$ that of sequential execution, while ideal weak scaling translates to constant execution time regardless of p.

Sequential and multi-threaded tests were conducted on the CompBio system at Georgia Institute of Technology. CompBio contains four 2.1GHz Intel Xeon E7-8870v3 processors with 45MB L3 cache, 1TB of DDR4 RAM, and four 4TB rotating disks in RAID 5 configuration. All tested software were compiled with GCC 5.3 and OpenMPI 1.10.2 if required. Distributed memory experiments were conducted on Iowa State University's CyEnce cluster. Each node contains two 2.0Ghz 8-core Intel Xeon E5-2650 CPUs and 128GB of RAM. The nodes are connected via quad data rate (QDR) Infiniband interconnect, and supported by a 288TB Lustre file system with one metadata server and 8 object storage servers. All data files are stored on Lustre with 1MB block size and stripe count of 8. All tested software were compiled with GCC 4.9.3 and MVAPICH 2.1.7. We repeated each experiment at least three times, and reported the fastest trials as they most closely reflect system capabilities.

5.1 K-mer Operations

Table 6: Time in seconds to reverse complement 1 million 31-mers using SIMD instructions.

	SEQ	SWAR	SSSE 3	AVX 2	AUTO
DNA	0.029	**0.0018**	0.0024	0.0036	*0.0019*
DNA5	0.049	0.0062	**0.0023**	0.0032	*0.0023*
IUPAC	0.049	0.0053	**0.0023**	0.0032	*0.0023*

We benchmarked the SIMD accelerated k-mer reverse complement operation using the CompBio system. Table 6 summarizes the time to reverse and complement 1 million DNA, DNA5, and IUPAC DNA 31-mers using the x86 SWAR, SSSE 3, and AVX 2 implementations. The "AUTO" implementation adaptively chooses the optimal instruction sets at compile time based on k-mer parameters. We achieved a throughput of approximately 2 microseconds per 31-mer. SWAR implementation was approximately 16× faster than sequential (SEQ in Table 6), while SSSE 3 was 21× faster. Speed up depends on k-mer length, with SWAR performing well for small k, and SSSE 3 the implementation of choice for larger k-mers. AVX 2, contrary to expectation, performed comparably to SSSE 3 instructions for k-mer reverse complements up to $k = 256$ (data not shown), due to the additional instructions required to copy bits between the 128 bit lanes.

5.2 Distributed File Reader

We benchmarked distributed file reading using the CyEnce cluster. To prevent file system cache from affecting the results, we used copies of the same file during each iteration of the test. Three different file access mechanisms were evaluated: MPI-IO, memory mapping (MMAP), and POSIX file access functions. Figure 4 shows that parallel reading of the R1 data set scaled nearly linearly up to $p = 64$, beyond which the network was likely saturated. MPI-IO and MMAP mechanisms performed similarly given CyEnce's configuration. For 32 and 64 processor cores, the POSIX mechanism showed an approximately 40% advantage.

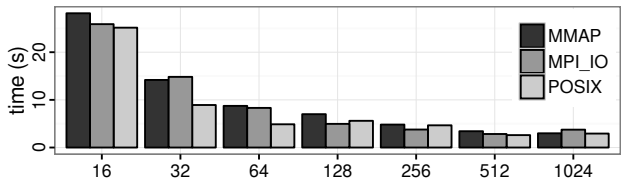


Figure 4: Time to read the R1 data set from disk into memory via MPI-IO, POSIX, and memory mapping, using varying number of processors. Only the x-axis is in logarithmic scale.

The effect of file system caching is dramatic. Table 7 shows the time to read the M4 data set using 128 processor cores with and without file caching. Caching had the most benefit for POSIX mechanism with a 22.8× speed up. The uncached file reading time suggests that saving and reading k-mer indices to and from disk likely is time consuming, as the network throughput to and from the parallel file systems becomes the limiting factor. Timing results from Sections 5.3 and 5.4 further suggest that rebuilding an index is likely preferable over loading a previously built index from disk.

Table 7: Time to read a file from the Lustre file system using 128 processor cores, with and without populated operating system file cache.

time (s)	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

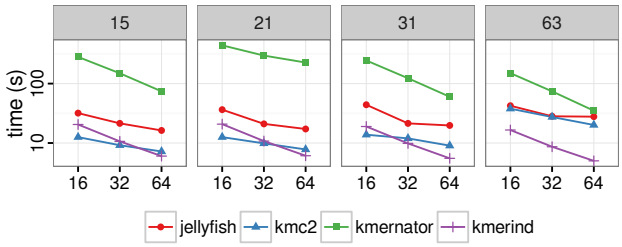
In subsequent tests, we used the POSIX mechanism for parallel file reading, and pre-populated the cache with a "warm up" iteration. For comparison with existing tools in Section 5.3, parallel file reading time was included for all tools, as KMC 2 did not measure it separately. For Kmerind scalability experiments in Section 5.4, the file reading time was excluded from the index construction and query times.

5.3 Comparisons with Existing Tools

We compared the performance of Kmerind to existing best-in-class k-mer counting tools on shared and distributed memory systems. JellyFish 2 [20] and KMC 2 [7] were chosen as they represent the most commonly used and fastest k-mer counting tools, respectively. Kmernator [16] was chosen as it was the only existing distributed k-mer counter.

5.3.1 Shared Memory Environment

We used the CompBio system for single node, multi-thread testing. For JellyFish 2 and KMC 2, we assigned one thread per core. For Kmerind, we treat the system as if it is a distributed memory system, assigning one MPI process per core. Kmernator was executed using MPI only as its performance worsened when multithreading was enabled in hybrid MPI-OpenMP mode. During execution, each thread or MPI process was locked to a single core using the *numactl* tool or via *mpirun*. All tools were configured to count canonical k-mers only and without any k-mer filtering.



time (s)	Varying k, 64 cores				$k = 31$, Varying cores		
	15	21	31	63	16	32	64
JellyFish 2	16.3	17.2	19.8	27.8	44.0	21.4	19.8
KMC 2	7.2	7.8	9.0	20.1	**13.8**	12.0	9.0
Kmernator	74.0	226.0	60.0	35.0	245.0	123.0	60.0
Kmerind	**6.0**	**6.1**	**5.5**	**5.0**	18.7	**9.9**	**5.5**

Figure 5: Strong scaling behavior for counting k-mers in R1 data set using 16, 32, and 64 cores on a shared memory system. Each plot corresponds to a different k value. The table shows the timing results for fixed core count p or fixed k

Strong scaling experiments were conducted with the R1 data set using 16, 32, and 64 cores. Figure 5 shows the performance of each tool or library for varying core counts p and k values. All tools showed performance improvement as p increased. In contrast to JellyFish 2 and KMC 2, both Kmerind and Kmernator showed nearly linear scaling, where the time approximately halved as p doubled. The performance of JellyFish 2 and KMC 2 decreased as k increased, suggesting inefficiencies in their k-mer parsing and comparison operations. The performance of Kmerind improved as k increased due to reduction in the number of valid k-mers in short input sequences as described in Section 3.1. For the R1 data set, the numbers of valid k-mers in millions are 2052, 1909, 1670, and 906 for k values of 15, 21, 31, and 63, respectively. Kmernator showed similar improvements, except when k was not near a power of 2 where we observed a significant performance penalty due to unknown cause.

Table 8: Time to count 31-mers in metagenomic read files and whole genomes using 64 cpu cores.

time (s)	Metagenomic Reads			Whole Genome	
	M1	M2	M3	G1	G2
JellyFish 2	36.27	67.35	84.46	20.60	66.13
KMC 2	23.44	48.19	83.28	115.63	341.35
Kmernator	84.00	172.00	349.00	–	–
Kmerind	**9.97**	**20.04**	**42.52**	**13.04**	**50.98**

Overall, Kmerind's performance was not strongly affected by values of k, scaled linearly with the number of cores, and consistently out-performed JellyFish 2 and Kmernator. Kmernator was slower than other tools by up to an order of magnitude. We observed that Kmerind performed worse than KMC 2 at low core count due to the larger message

size $|M|/p$ increasing Kmerind's synchronous communication overhead. At higher core count, KMC 2's performance degraded, likely due to high thread synchronization overhead. We expect the performance gap between KMC 2 and Kmerind to widen in Kmerind's favor as core count is increased further. The table in Figure 5 shows the time for index construction for all tools with k or p fixed. Kmerind was up to 1.64× faster than KMC 2 for counting canonical 31-mers using 64 cores, and 3.6× faster than JellyFish 2.

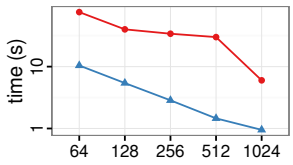
Table 8 shows the performance of JellyFish 2, KMC 2, Kmernator, and Kmerind for varying data sizes. The Metagenomic Reads experiments were conducted with M1, M2, and M3 data sets using $p = 64$ and $k = 31$. All tools exhibited near linear scaling with data size. Kmerind counted the M3 data set in less than 43 seconds and was 2 to 2.4 times faster than KMC 2 for all data sets. The Whole Genome experiments were conducted with G1 and G2 data sets. Kmerind completed counting the human genome in 13 seconds and pine genome in 51 seconds using 64 cores. JellyFish 2 was 1.6 and 1.3 times slower than Kmerind for G1 and G2, while KMC 2 was up to 9 times slower. Kmernator was excluded from the experiment due to its lack of FASTA file support.

Table 9: Time to query 1% of indexed k-mers with JellyFish 2 and Kmerind.

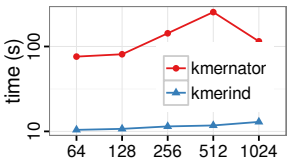
	Varying k, 1 Core			
time (s)	15	21	31	63
JellyFish 2	0.66	1.284	0.821	2.074
Kmerind	0.09	0.10	0.11	0.20

Of the three existing tools tested, only JellyFish 2 provides a command line interface to query the database. KMC 2 does provide an option to find intersection between two databases, but outputs the minimum counts for the entries in the intersection. Table 9 shows the times to query JellyFish 2 and Kmerind count indices using 1% of indexed k-mers on CompBio using a single core, as JellyFish 2 only supports single threaded queries. As JellyFish 2 requires loading the database file from disk, results are not directly comparable but is illustrative of the benefit of Kmerind's in-memory index for online queries.

5.3.2 *Distributed Memory Environment*



(a) strong scaling



(b) weak scaling

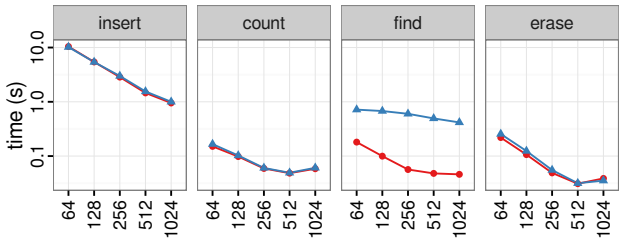
		Core Count				
	time (s)	64	128	256	512	1024
(a)	Kmernator	76.00	40.00	34.00	30.00	6.00
	Kmerind	**10.43**	**5.42**	**2.85**	**1.46**	**0.95**
(b)	Kmernator	76.00	81.00	143.00	254.00	115.00
	Kmerind	**10.43**	**10.71**	**11.45**	**11.75**	**12.94**

Figure 6: Strong and weak scaling behaviors in Kmerind and Kmernator for distributed DNA 31-mer counting. Data set M1 was used for strong scaling while sets M1–M5 were used for weak scaling.

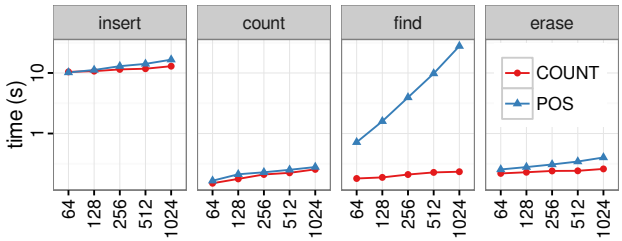
We benchmarked index construction for Kmerind and Kmernator using data sets M1–M5 and 64 to 1024 processor cores on CyEnce for Kmerind and Kmernator. Figure 6 shows Kmerind was consistently faster than Kmernator by at least a factor of 6 for strong scaling and 8 for weak scaling. Kmerind's showed approximately linear strong scaling for up to 512 processor cores, beyond which the parallel efficiency decreased slightly. For weak scaling, Kmerind 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. Kmernator showed a reproducible non-linear scaling behaviors for 256 and 512 processor cores due to unknown cause.

5.4 Scalability

Kmerind's indices are general and extend beyond just counting. Figure 7 shows strong and weak scaling behavior of Kmerind's count and position indices for each index operation. The *count*, *find*, and *erase* operations used 1% of the indexed k-mers, selected randomly, as input. All experiments were performed using data sets M1–M5 on CyEnce.



(a) strong scaling

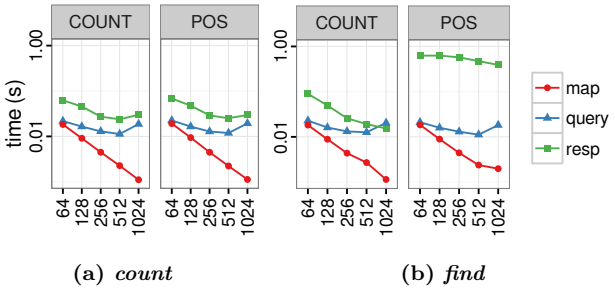


(b) weak scaling

			Core Count				
		time (s)	64	128	256	512	1024
(a)	Count Index	insert	10.43	5.42	2.85	1.46	0.95
		find	0.18	0.10	0.06	0.05	0.05
	Position Index	insert	10.19	5.40	2.99	1.55	1.00
		count	0.17	0.10	0.06	0.05	0.06
		find	0.72	0.68	0.60	0.49	0.42
(b)	Count Index	insert	10.43	10.71	11.45	11.75	12.94
		find	0.18	0.19	0.21	0.23	0.23
	Position Index	insert	10.19	11.20	12.95	14.12	16.57
		count	0.17	0.21	0.23	0.25	0.28
		find	0.72	1.59	3.94	9.79	27.77

Figure 7: Strong and weak scaling result for the *insert*, *count*, *find*, and *erase* operations for Kmerind's count and position indices, using DNA 31-mers from data sets M1–M5.

Kmerind count and position indices ingested the M1 data set in approximately one second, and the M5 data set in 13 and 16.6 seconds respectively using 1024 processor cores. Retrieving the counts in the count index required 0.05 and 0.23 seconds for M1 and M5, respectively, while counting using position index required slightly more time. Retrieving the positions took 0.42 and 28 seconds for M1 and M5.

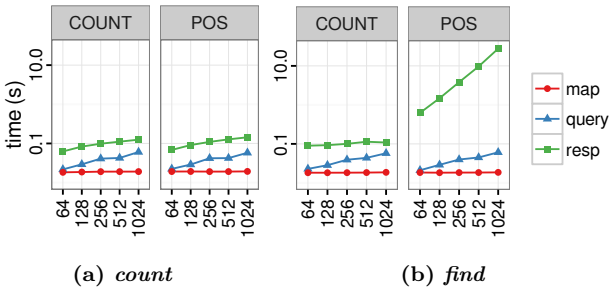


(a) *count* (b) *find*

Figure 8: Strong scaling of internal steps in the *count* and *find* operations for count and position indices. The "map", and "query" steps correspond to the *map_to_process* and *distribute* functions in Algorithms 1 and 2, while "resp" corresponds to all remaining algorithmic steps after the *distribute* step.

In both strong and weak experiments, the *insert*, *count*, and *erase* operations behaved similarly between the count and position indices, while the *find* operation diverged between them, reflecting the algorithmic and complexity differences described in Sections 3.3 and 3.4. In strong scaling experiments, *insert* showed near linear scaling, as expected, while *count* and *erase* reached minima at 512 processor cores. Figure 8a shows that the presence of the minima is largely due to communication in the "query" and "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 (Figure 8b) with complexity $(r|M|/p) + \tau p + \mu(r|M|/p)$ (Section 3.4). 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 in a processor can introduce load imbalance that further causes the run time to scale sub-optimally.



(a) *count* (b) *find*

Figure 9: Weak scaling of internal steps in the *count* and *find* operations for count and position indices. The "map", "query", and "resp" steps are defined identically as those in Figure 8.

Weak scaling experiments showed a slight increase of run time as p increased for *insert*, *count*, and *erase* operations (Figure 7b). Figure 9a shows that this is due primarily to the "query" step and to a lesser degree the "resp" 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 increases with $\log(p)$. The time for the *find* operation in position index increased linearly with p and r. As the expected k-mer frequency scales with p since the data size is kept constant per processor for weak scaling experiments, the *find* operation complexity $\tau p + (\mu + 1)(r|M_i|)$ (Section 3.4) scales linearly with p, as evident in the "resp" step scaling in Figure 9b and in the average repeat rates of data sets M1–M5, which are 1.11, 1.17, 1.26, 1.37, and 1.54 respectively.

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. Consequently, it has become increasingly critical to develop k-mer counting and indexing tools and 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 be stored 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 is competitive with current best-in-class k-mer counting tools at low core counts and out-performs them at high core counts.

While the library was implemented using distributed memory parallel algorithms, Kmerind's API has been designed with sequential semantics to facilitate application development. The API is 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, minimize local computation and memory utilization, or to integrate novel indexing strategies. For example, a Bloom filter can be used, similar to the approach taken by JellyFish 2, in a preprocessing step to minimize number of k-mer indexed, provided that singleton k-mer exclusion is compatible with the application requirements.

7. ACKNOWLEDGMENTS

This research is supported in part by the National Science Foundation under IIS-1416259, CNS-1229081, and an Intel Parallel Computing Center award. *Conflict of interest:* none declared.

8. REFERENCES

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, Mar. 2004.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, Oct. 1990.

[3] P. Audano and F. Vannberg. KAnalyze: A Fast Versatile Pipelined K-mer Toolkit. *Bioinformatics*, page btu152, Mar. 2014.

[4] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov, and E. S. Lander. ARACHNE: A Whole-Genome Shotgun Assembler. *Genome Research*, 12(1):177–189, Jan. 2002.

[5] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[6] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-min Sketch and Its Applications. *J. Algorithms*, 55(1):58–75, Apr. 2005.

[7] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, May 2015.

[8] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, 2000. Proceedings*, pages 390–398, 2000.

[9] R. J. Fisher, A. J. Fisher, and H. G. Dietz. Compiling For SIMD Within A Register. In *11th Annual Workshop on Languages and Compilers for Parallel Computing*, LCPC '98, pages 290–304. Springer Verlag, 1998.

[10] P. Flick. mxx. `http://patflick.github.io/mxx/`.

[11] P. Flick and S. Aluru. Parallel Distributed Memory Construction of Suffix and Longest Common Prefix Arrays. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 16:1–16:10. ACM, 2015.

[12] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick. HipMer: An Extreme-scale De Novo Genome Assembler. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 14:1–14:11. ACM, 2015.

[13] Google SparseHash. `https://github.com/sparsehash/sparsehash`.

[14] D. R. Kelley, M. C. Schatz, and S. L. Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, 11(11):1–13, 2010.

[15] W. J. Kent. BLAT–the BLAST-like alignment tool. *Genome Research*, 12(4):656–664, Apr. 2002.

[16] Kmernator. `https://github.com/JGI-Bioinformatics/Kmernator`.

[17] 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*, 9:517, 2008.

[18] Y. Li and XifengYan. MSPKmerCounter: A Fast and Memory Efficient Approach for K-mer Counting. *arXiv:1505.06550 [cs, q-bio]*, May 2015. arXiv: 1505.06550.

[19] Y. Liu, J. Schröder, and B. Schmidt. Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics*, 29(3):308–315, 2013.

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

[21] P. Melsted and J. K. Pritchard. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, 12:333, 2011.

[22] B. Nystedt *et. al.* The Norway spruce genome sequence and conifer genome evolution. *Nature*, 497(7451):579–584, May 2013.

[23] N. Philippe, M. Salson, T. Lecroq, M. Léonard, T. Commes, and E. Rivals. Querying large read collections in main memory: a versatile data structure. *BMC Bioinformatics*, 12(1):242, June 2011.

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

[25] R. S. Roy, D. Bhattacharya, and A. Schliep. Turtle: Identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, 30(14):1950–1957, July 2014.

[26] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. M. Jones, and Ä. Birol. ABySS: A parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, June 2009.

[27] N. Välimäki and E. Rivals. Scalable and Versatile k-mer Indexing for High-Throughput Sequencing Data. In *Bioinformatics Research and Applications*, number 7875 in Lecture Notes in Computer Science, pages 237–248. Springer, May 2013. DOI: 10.1007/978-3-642-38036-5_24.

[28] X. Yang, K. S. Dorman, and S. Aluru. Reptile: representative tiling for short read error correction. *Bioinformatics (Oxford, England)*, 26(20):2526–2533, Oct. 2010.

[29] D. R. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(5):821–829, May 2008.

[30] 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*, 9(7):e101271, July 2014.