Distributed-Memory k-mer Counting on GPUs

Israt Nisa[†], Prashant Pandey[*†], Marquita Ellis [*†], Leonid Oliker[†], Aydın Buluç[*†] Katherine Yelick[*†]
[*]Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
[†]Computational Research Division, Lawrence Berkeley National Laboratory

***Abstract*—A fundamental step in many bioinformatics computations is to count the frequency of fixed-length sequences, called k-mers, a problem that has received considerable attention as an important target for shared memory parallelization. With datasets growing at an exponential rate, distributed memory parallelization is becoming increasingly critical. Existing distributed memory k-mer counters do not take advantage of GPUs for accelerating computations. Additionally, they do not employ domain-specific optimizations to reduce communication volume in a distributed environment. In this paper, we present the first GPU-accelerated distributed-memory parallel k-mer counter. We evaluate the communication volume as the major bottleneck in scaling k-mer counting to multiple GPU-equipped compute nodes and implement a supermer-based optimization to reduce the communication volume and to enhance scalability. Our empirical analysis examines the balance of communication to computation on a state-of-the-art system, the Summit supercomputer at Oak Ridge National Lab. Results show overall speedups of up to two orders of magnitude with GPU optimization over CPU-based k-mer counters. Furthermore, we show an additional $1.5\times$ speedup using the supermer-based communication optimization.**

***Index Terms*—GPU, k-mer counter, distributed memory**

I. Introduction

A significant number of *omics* applications, including genomics, transcriptomics, metagenomics, and proteomics, rely on efficient counting of length-k subsequences called k-mers. This step, which often takes a large fraction of the overall application runtime, is a key computation within popular tools that perform taxonomic assignment [32], metagenome classification [3], genome and metagenome assembly [15], and transcriptome abundance estimation [26].

Due to lowering costs and exponential growth of sequence data, the input sets for k-mer counting analysis are exceeding the memory capacities of single shared memory systems, even high end servers. This results in a performance penalty for software that needs to rely on external memory. A growing number of tools from the bioinformatics community try to address this issue by compressing k-mers [20], offering more space-efficient data structures [24], or using external memory [8] during k-mer counting.

In addition to space constraints, computational costs on a single workstation can make the processing requirements of k-mer counting on modern data sets impractical. To reduce this run time, a recent work Gerbil [8] has shown that GPUs can be a superior alternative to CPUs for computing k-mer counts. Unfortunately, GPUs generally have smaller memories compared to CPUs, making single GPU nodes less practical for analyzing larger data sets.

In this work, we address these challenges by presenting the first k-mer counting methodology that leverages the computational power and scalable memory capacity of distributed-memory GPU systems. Results using real, large-scale, genomic data sets demonstrate that our GPU-based computation significantly reduces the execution time by one to two orders of magnitude, compared to its distributed-memory CPU counterpart used in a long-read aligner [7]. However, the GPU accelerated kernels make communication the new bottleneck, as explained in section III-C. Note that, the communication volume is independent of the CPU- or GPU-based counting.

To address the communication bottleneck, we partition the input data using minimizers. Minimizers allow efficient binning of k-mers such that sequences with overlaps are binned together [27], [28]. Using the minimizers, we partition data into supermers which are larger than k-mers in length, and distribute supermers directly across nodes thereby reducing the communication volume. Our supermer-based partitioning is independent of the GPU implementation and can be used in other distributed-memory k-mer counters to reduce the communication volume.

The key contributions of out work include:

- We propose a high-performance distributed k-mer counter for GPUs, as an extension of the state-of-the-art k-mer counter for distributed-memory. Experimental results demonstrate one to two orders of magnitude improvement across a variety of genomic data sets at scale.
- To address the inter-node communication bottleneck that is exacerbated by the distributed implementation, our novel approach leverages *supermers* to reduce communication volume by up to $4\times$, which is supported by an analysis of its theoretical communication reduction potential.
- Overall, our GPU accelerated supermer-based k-mer counter on the Summit system outperforms the existing CPU based framework by up to $150\times$ on the *H. sapien* 54X genome data set, and opens the door to *omics* computations at unprecedented scale.

The rest of the paper is organized around these optimizations, as follows: Section II gives an overview of the k-mer counter and introduces the notion of *minimizers* which are key to the supermer approach. Section III describes our proposed GPU-based k-mer counter, and section IV discusses the optimization technique to reduce communication using supermers. Section V presents experimental evaluation on high end multicore and GPU system, with up to 128 nodes with either 6 GPUs or 42 CPU cores per node. Our analysis also explores some of the trade-offs in the design space and identifies the scaling bottlenecks with each version. Section VI discusses the related work, and Section VII concludes.

II. Background

A. k-mer counting

k-mer counting is the process of counting length-k subsequences—strings of k nucleotides or bases (A, C, T, G)—in genomic or transcriptomic data. Many analyses in computational biology begin by counting k-mer occurrences in the input data [3], [15], [26], [32]. The resulting k-mer histograms are valuable for understanding the distributions of genomic subsequences, creating "profiles" of genome and metagenomic data, identifying k-mers of scientific interest by frequency, and so on. These histograms support many downstream analyses as well. For example, as a (weighted) de Bruijn graph representation [4], [11], [25] , as an input for building large-scale sequence search indexes [2], [23], [29]–[31], and for weighted locality sensitive hashing [18].

k-mer counting is one of the most common and arguably one of the simplest preliminary steps in many bioinformatics analyses. The number of existing papers on this problem suggests, however, that efficient execution of this task, with reasonable memory use, is far from trivial. k-mer counting is non-trivial because the datasets can exceed single node shared memory resources and the frequency distribution of the k-mers is often skewed. There are many different k-mer counter software implementations, because there are many different competing performance issues, including space consumption, cache-locality, scalability with multiple threads, and in a distributed memory setting, with multiple nodes.

Irregularity in the input data makes k-mer counting a challenging problem for distributed memory parallelization. In particular, the distribution of k-mers is not fixed across biological input datasets and cannot be determined until the run time. The primary methods for scalable-distributed memory k-mer counting rely on distributed hash tables [6], [7], [10], [12], [21], [33].

B. Minimizers

A *minimizer* [27], [28] of a k-mer is the smallest subsequence of length m $(m < k)$ based on an arbitrary order. In genomic sequences, consecutive k-mers often share the same minimizer and can be represented as a single sequence of more than k bases, significantly reducing the redundancy. Minimizers have been extensively used to efficiently process and represent genomic sequences [1], [4], [5], [16], [27], [28], [32].

In the context of k-mer counting, minimizers have been used to partition genomic data into bins and these bins are then processed sequentially and independently for k-mer counting reducing the maximum main memory requirements.

Algorithm 1 Distributed k-mer counting [7]

```
1:  procedure ParseKmer                    ▷ Read files and parse k-mers
2:      for $r \in reads$ do
3:          for $i = 0$ to Len($r$) $- k + 1$ do
4:              $kmer$ = ExtractKmer($r, i, k$)
5:              $P$ = Hash($kmer, nProc$)         ▷ Find processor
6:              $kmers[P] \leftarrow kmer$        ▷ Outgoing kmer vector

7:  procedure ExchangeKmer                  ▷ Communicate k-mers
8:      $recvdkmers$ = Alltoallv($kmers, P$)

9:  procedure CountKmer                     ▷ Count local k-mers
10:     $kcounter \leftarrow$ Initialize
11:     for $kmer \in recvdkmers$ do
12:         if $kcounter$.Has($kmer$) then
13:             $kcounter$.Increment($kmer$)
14:         else
15:             $kcounter$.Insert($kmer$)
```

Partitioning the k-mers based on minimizers ensures that a k-mer, irrespective of the its location in the genomic sequence, will always end up in the same bin.

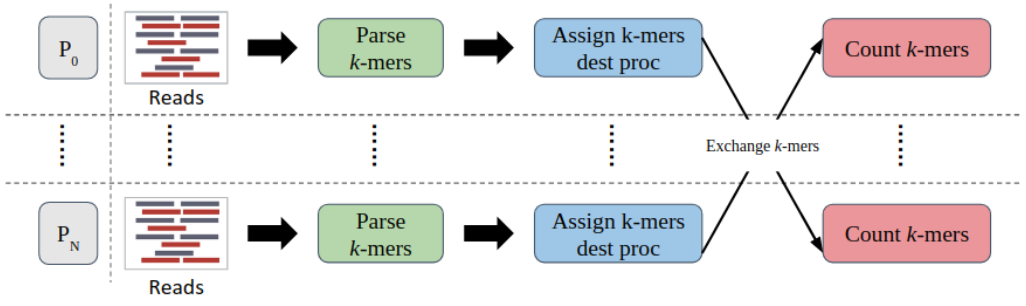
Multiple minimizer orderings have been proposed in different contexts. The simplest to reason about pecind implement is the lexicographical ordering proposed by Roberts [27]. However, in practice, the lexicographical ordering often leads to unbalanced partitions. This leads to performance being dominated by the largest bin. The KMC2 [5] k-mer counter, introduced a slight modification in lexicographical ordering by assigning lower priority to m-mers starting with AAA or ACA. This special ordering helps to spread out k-mers and reduces the skew in the bin sizes. This ordering is also used by GPU-based k-mer counter Gerbil [8].

III. Distributed-Memory k-mer counter on GPUs

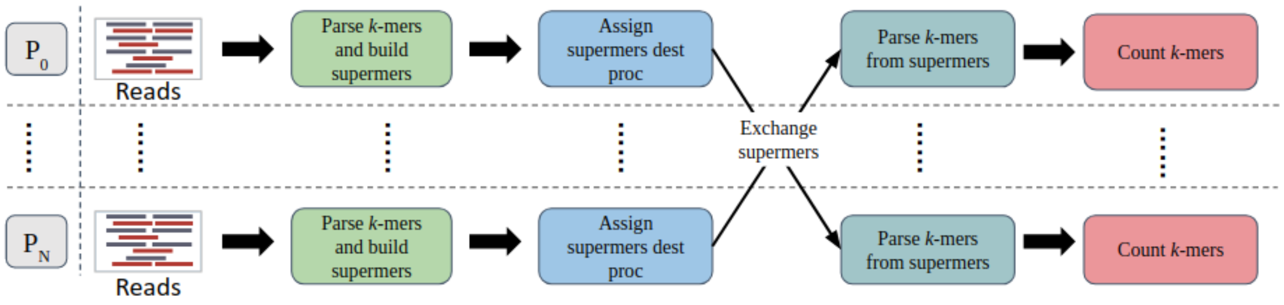
The proposed GPU based distributed memory k-mer counter in this work follows the general approach introduced by Georganas et al. [12], originally developed for CPUs. On a high level, the framework is consist of three main modules: 1)parse & process k-mers- parses the reads into k-mers and finds destination for them, 2)exchange the k-mers to their corresponding destination, and 3)build the k-mer counter at each local machine. Figure 1a demonstrates the main modules of the distributed k-mer counting pipeline.

A. k-mer counter on CPU [12]

In the CPU based framework by [12], also as shown in Figure 1a, at the beginning, input reads are partitioned across processors. Next, independently in parallel, processors parse k-mers from input reads and map the k-mers to a partition of the global hash table (and "owner" processor) via a uniform-random hash function. The choice of the hash function is critical, but does not alone guarantee load balance — an issue addressed in subsequent discussion. The end result is



(a) Pipeline in a distributed k-mer counter.


P0
Reads
Parse k-mers and build supermers
Assign supermers dest proc
Parse k-mers from supermers
Count k-mers
Exchange supermers
PN
Reads
Parse k-mers and build supermers
Assign supermers dest proc
Parse k-mers from supermers
Count k-mers

(b) Pipeline in a distributed k-mer counter on GPUs.

Fig. 1: k-mer counting pipeline in a distributed environment.

a partitioned hash table storing the set of all k-mers and their respective global frequencies.

Algorithm 1 demonstrates an overview of this approach. Line 2 iterates over the reads from a given read partition and Line 3 iterates over the bases in the read. To construct the global hash table, all instances of a k-mer are mapped to a single hash table location on the same processor. Line 5 uses MurmurHash3 hash function to find the destination processor. Any given instance of a k-mer will be communicated from the source processor to a single processor owning the respective hash table partition. This many-to-many exchange is implemented using MPI Alltoall and Alltoallv routines, as shown in the Line 8.The k-mers are stored and counted in the destinations' partition of the global hash table[1] as shown from Line 11 to Line 15. Depending on the total size of the input, relative to software limits (approximating available memory), the computation and communication may proceed in multiple rounds.

B. *k-mer counter on GPU*

1) Parse & process k-mers: In the GPU-based solution, as a first step, we concatenate the input reads into one long array of bases and mark the read ends by special bases, before copying the data to GPU memory. Inside the GPU kernel, the copied array is evenly partitioned into smaller chunks of bases and is assigned to different thread blocks (group of threads/warps). In a genomic dataset, individual reads from the

[1]See also the analysis of destination- versus source-side k-mer consolidation by Georganas [10].

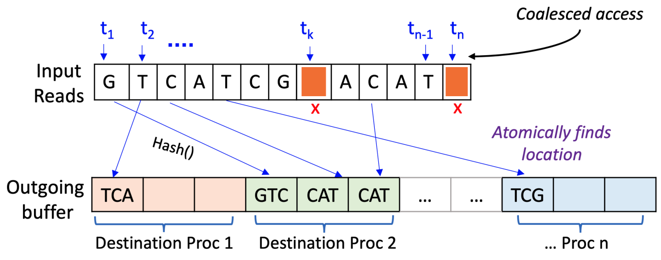
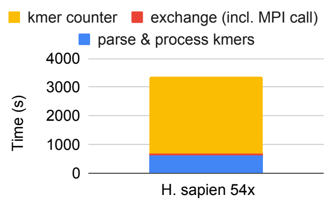


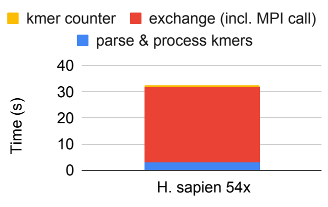
Fig. 2: Parsing k-mers (k=3) from reads and mapping k-mers to destinations processors using k-mer based k-mer counter on GPU

same read partition can have a big variance in their lengths. Moreover, the performance on GPUs is highly sensitive to load imbalance across threads, warps (group of 32 threads), or thread-blocks (group of warps). This even work distribution provides a balanced work load and avoids the variance coming from the lengths of reads.

To exploit coalesced memory accesses on GPU (combining multiple memory accesses into a single transaction), which is another crucial factor for high performance, consecutive threads are mapped to a continuous series of bases. Figure 2 shows the mapping of the bases and the work distribution across threads. The threads starts constructing the k-mer starting from the base and reads up to k-mer length number of bases. Thread t_1 and t_2 start reading at G and T, and build k-mer GTC and TCA, respectively. This technique also ensures



(a) 2688 CPU cores



(b) 384 GPUs

Fig. 3: Runtime breakdown of CPU- and GPU-based k-mer counters on 64 nodes for the *H. sapien* 54X dataset. Note that due to k-mer counting GPU acceleration, the y-axis in (a) is two orders of magnitude higher than (b). The k-mer exchange time is roughly the same across (a) and (b).

intra-warp load balance.

After parsing k-mers, a thread finds the destination processor of each k-mer using the hash function, as mentioned in the previous paragraph. A global array, *outgoing_buffer*, with P partitions, as shown in fig. 2 stores the k-mers in their corresponding destinations (outgoing processor). The numbers of k-mers at the partitions can vary depending on the skewness of the datasets. As all the GPU threads concurrently update this buffer, the update operation is performed atomically. In genomic or transcriptomic sequences, we encounter four different bases A, C, G, T, which can be encoded into 2-bit representations and compress the representation of the k-mers [7]. For example, a p11-mer k-mer can fit into a 32 bit data type instead of an $11 \cdot 8 = 88$ bit character array.

2) Exchange k-mers: In the next step, the k-mers are communicated between processes via MPI_Alltoall and MPI_Alltoallv routines. Depending on the underlying connection of the system, we can deploy a GPUDirect communication, where data can be directly transferred between GPUs. Alternatively, a CPU based communication can be used, where data are first copied to CPU, which handles the communication, and finally, copied back to GPU. Our current framework supports both methods.

3) Build k-mer counter: In the last stage of the pipeline, we maintain a global k-mer counter where each unique k-mer is stored along with its frequency. To count the received k-mers, the same thread mapping strategy is used where each thread processes one k-mer. k-mers find unique slots in the k-mer counter table via the MurmurHash3 function. If the table has already seen the k-mer, it increments its frequency, otherwise, a new k-mer is inserted. Both operations are handled atomically to avoid race conditions. Collisions are addressed using similar concept as the open-addressing based hash table. If a slot the k-mer is hashed to is already occupied by another k-mer, it seeks for a free slot in a probe sequence (linear, quadratic, etc). In this work, we use linear probing to find the next empty slot.

C. GPU acceleration and communication bottleneck

Our GPU-acceleration of k-mer counting provides a significant computational and performance improvement over CPU-based k-mer counting. Figures 3a and 3b show a breakdown of runtime, on the *H. sapien* 54X dataset (Table I), of the CPU baseline version alongside the GPU accelerated version using 64 Summit nodes. The CPU baseline was gathered using 42 cores (IBM Power9) per node and the GPU accelerated data use 6 GPUs (NVIDIA V100) per node. More details on the Summit cluster can be found in section V-A. For this important computation, our results show an impressive GPU code acceleration of $100\times$ compared to the CPU baseline, a reduction in overall runtime from approximately 50 minutes to just 30 seconds (excluding I/O).

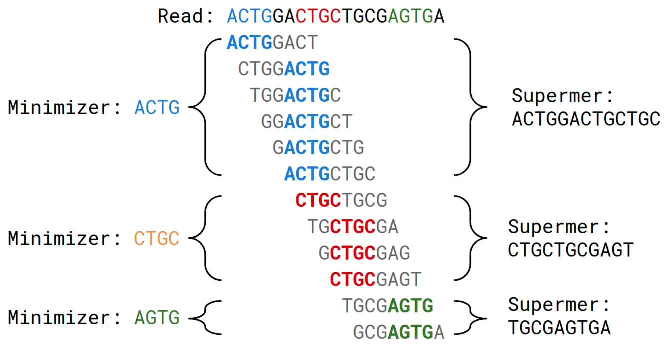


Fig. 4: Parsing a read for supermers with k-mer length 8 and minimizer length 4. We are not cannonicalizing the k-mers and minimizers. For simplicity, we use the lexicographical order to choose the smallest minimizer.

Due to this speedup, the computational overhead of the k-mer counter has effectively been minimized via our GPU optimization. As highlighted in Figure 3b, the new bottleneck now becomes communication, primarily the many-to-many exchange of k-mers implemented with MPI_Alltoallv. This communication cost can hinder the distributed memory scalability of the GPU-accelerated k-mer counters. In the next section, we address this challenge by leveraging a domain-specific approach in data packing to reduce data exchange overhead and facilitate scalability.

IV. Optimizing communication

There are two main approaches that we use to reduce the communication cost in our GPU-accelerated k-mer counter. First, we use a minimizer-based scheme to partition and transfer sequencing data to GPU nodes for k-mer counting. Second, instead of sending individual k-mers across nodes for counting we send sequences longer than k bases.

A. Supermers

A *supermer* is a contiguous sequence of bases wherein each k-mer shares the same minimizer. During the read parsing phase, instead of computing individual k-mers using a sliding window of length k, we extend the window until we find a k-mer with a different minimizer. These supermers are then sent to their respective nodes to for k-mer counting.

The supermers are generally longer than k-mers and help in reducing the data amount that is required to be communicated across nodes. Figure 4 shows an example of how to compute supermers from a given read. The read is of length 19 bases, k-mer length 8, and minimizer length 4. We use lexicographical ordering to choose the smallest minimizer in each k-mer. In the traditional setting, parsing k-mers from the read and sending k-mers to the respective GPU nodes for counting would require $(19-8+1)\cdot 8 = 96$ bases to be communicated. However, our approach only requires three supermers of total length 33 (average length 11 per supermer) bases, which results in a total communication reduction of $2.9\times$.

To communicate, we partition the supermers based on the minimizers and assign a single GPU node to each minimizer for counting. The minimizer-based partitioning guarantees that each k-mer, irrespective of the supermer it appears in, will always be sent to the same node for counting. The minimizer-based partitioning scheme has been previously applied in k-mer counting tools to reduce the main memory requirements and speed up the counting process [5]. Gerbil k-mer counter [8] also uses the supermer approach to partition the data into temporary files such that all occurrences of a certain k-mer are together. In this work, we extend the concept of minimizers and supermers to efficiently distribute data across multiple processors.

Choosing the minimizer based on the lexicographical ordering produces skewed or imbalanced partitions [4], [5]. In the past, people have come up with custom orderings for minimizers that are optimized for the underlying genomic data. However, computing custom orderings during supermer construction incurs extra computational overhead. An easier way to break away from lexicographical ordering without having to compute a custom ordering is to map nucleotide bases A, C, T, G to 2-bit representations in a random order. Specifically, we map $A = 1$, $C = 0$, $T = 2$, $G = 3$. This random ordering of bases implicitly creates a custom ordering for minimizers and tends to spread out supermers and produce balanced partitions. This ordering have been explored by Squeakr [24] in the past for k-mer counting.

B. *Using window to build supermers on GPUs*

In this section, we discuss the challenges of extending the supermer concept to a massively parallel architecture like GPUs. In an ideal scenario, a complete read can be mapped to a thread, and the thread can sequentially parse the k-mers, find the minimizers, and extend the supermer as long as it shares a common minimizer with the previous k-mer. In the k-mer-based k-mer counter, coalesced memory access and low variance in workload distribution are achieved by processing the k-mers independently in parallel. In the supermer-based case, multiple threads need to communicate with each other to check if they have a common minimizer and can contribute to the same supermer. To avoid thread communication as well as write conflict, we partition reads into smaller windows and assign one thread to process all the k-mers in that window. Therefore, a thread uses its private register to update the supermer and writes out the final version in the global memory. Figure 5 demonstrates an example with a window length

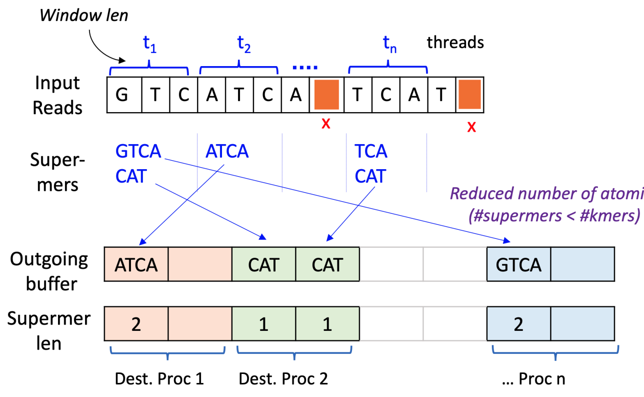


Fig. 5: Parsing k-mers (k=3) from reads, building supermers, and mapping supermers to destinations processors using $supermer$ based k-mer counter on GPU

of three. Thread t_1, at first, processes k-mer GTC, finds its minimizer (e.g., TC), and starts constructing a supermer starting from GTC. Then, it moves on the second k-mer TCA, and follows the same process. As they share same minimizer, e.g., TC, it extends the supermer to $GTC-A$. Each thread processes all the k-mers from the same window. This technique also helps to resolve the load imbalance issue coming from variation in length of reads.

However, by partitioning the reads into windows, we limit the length of the supermers. The maximum length of the supermer can only be as long as the length of the window. In this work, we select a window size depending on the length of k-mer (user provided) and efficiency of packing supermers into one or more machine words. An extra buffer is also maintained to store the length of each supermer (i.e., the number of k-mers packed inside).

After constructing the supermers, similar to k-mer based framework from Section III, each thread finds the destination processor based on the minimizer of the supermer and populates the *outgoing_buffer* with supermers. Next, the supermer array and the length of the supermers are communicated via MPI_AlltoAllv among processors. At the receiving end of the processors, each supermer is parsed into its constituent k-mers and the k-mers are counted accordingly.

Results show that the supermer-based version increases runtime by an average of 27% due to the overhead of constructing the supermers and 23% for counting the k-mers (compared to the k-mer-only version). However, given that the GPU-based k-mer counter is a communication-bound problem, the supermer approach achieves an overall performance improvement as shown in Section V.

C. *Supermer implementation*

Algorithm 2 shows the pseudo code for computing supermers given a read and minimizer and window sizes. Each read is processed in chunks of window size w (Line 3). In our implementation, the window length was set to 15 (for a k-mer length of 17) in order to be able to encode each supermer in

Algorithm 2 Supermer–based distributed k-mer counting

```
1:  procedure BuildSupermer                              ▷ Parse k-mers and build supermer
2:      for $r \in reads$ do
3:          for $i = 0$ to $\text{Len}(r) - k + 1$ step $window$ do
4:              $kmer = \text{ExtractKmer}(r, i, k)$
5:              $minimizer = \text{Minimizer}(kmer)$
6:              $prev \leftarrow minimizer$
7:              $P = \text{Hash}(minimizer, nProc)$                ▷ Find processor
8:              $supermers[P].\text{PushBack}(kmer)$
9:              $slens[P].\text{PushBack}(k)$
10:             $\text{Increment}(nSmer[P])$
11:             for $w = 1$ to $window$ do
12:                 $kmer = \text{ExtractKmer}(r, i + w, k)$
13:                 $minimizer = \text{Minimizer}(kmer)$
14:                 if $minimizer \neq prev$ then
15:                     $P = \text{Hash}(minimizer, nProc)$
16:                     $supermers[P].\text{PushBack}(kmer)$
17:                     $slens[P].\text{PushBack}(k)$
18:                     $\text{Increment}(nSmer[P])$
19:                 else
20:                     $supermers[P][nSmer].\text{AddChar}(kmer, k - 1)$
21:                     $\text{Increment}(slen[P][nSmer[P]])$
22:                 $prev \leftarrow minimizer$
23: procedure ExchangeSupermer
24:     $recvdsupermers = \text{AlltoallV}(supermers, P)$
25:     $recvdlens = \text{AlltoallV}(slens, P)$
26: procedure CountKmer                                  ▷ Count local k-mers
27:     $kcounter \leftarrow \text{Initialize}$
28:     for $supermer \in recvdsupermers$ do
29:         for $slen \in recvdlens$ do
30:             $kmer = supermer.\text{ExtractKmer}(slen, k)$
31:             if $kcounter.\text{Has}(kmer)$ then
32:                 $kcounter.\text{Increment}(kmer)$
33:             else
34:                 $kcounter.\text{Insert}(kmer)$
```

a single 64-bit machine word. For each window, k-mers are appended to the same supermer sequence until a k-mer with a different minimizer is found (Line 14). Each supermer is assigned to a processor for counting based on the hash value of the minimizer (Line 7 and 15). All the supermers, and thereby all the k-mers that share the same minimizer, are always assigned to the same processor for counting. These supermers are stored in an array corresponding to each processor along with the lengths of those supermers. During counting, k-mers are first extracted from the supermers and then counting is performed using a hash table.

D. Theoretical analysis

We now perform a theoretical analysis to determine the communication volume in the GPU-accelerated k-mer counter. We analyze both k-mer- and supermer-based approaches. We will use the following notation in the communication volume analysis:

m	length of minimizer
k	length of k-mer
w	window to select supermers
s	Average length of supermers
D	Total input data size
P	Number of parallel processors
L	Average length of reads

We begin by assuming the input of size D is partitioned roughly uniformly over P parallel processors. This is ensured by the parallel I/O in the implementation. Let the average length of reads in the input be L. So there are D/L input reads in total, and $L - k + 1$ k-mers are computed from each read on average. The total size of the k-mer multiset is therefore $K \approx \frac{D}{L}(L - k + 1)$. With a uniform distribution of the input, the computational cost of computing k-mers from the reads in parallel is $O(K/P)$. Given a hash function that maps k-mers to processor partitions uniformly at random, each processor will communicate $O(\frac{P-1}{P} \times \frac{K}{P})$ k-mers to other processors. The per processor communication volume is therefore, $O(\frac{P-1}{P} \times \frac{K}{P} \times k)$.

In the supermer-based k-mer counter, nodes communicate supermers instead of k-mers. If the average length of a supermer is s ($s > k$) then total size of supermers across all reads would be $S \approx \frac{D}{L}(L - s + 1)$. Therefore, the total volume of communication would be $\mathcal{O}(\frac{P-1}{P} \times \frac{S}{P} \times s)$. Given that the average length of supermers is dependent on the characteristics of the input reads, it is hard to come up with an exact communication bound using the variables involved in computing supermers. However, if the average supermer length is s and k-mer length is k, then the total reduction in communication is $\approx (s - k)\times$. As explained in section IV-A, where the k-mer length is 8 and average supermer length is 11 the total reduction is $2.90\times$.

V. Results and Discussion

A. Experiment setup

We present performance results of our k-mer and supermer-based k-mer counters with GPU acceleration. Source codes are available for download[2]. As a performance baseline, we employ a CPU-only k-mer counter that is the same as our k-mer counter without the GPU acceleration and supermer optimizations. This CPU-only version was derived from the k-mer analysis component of diBELLA [7], utilizing only the k-mer counting features.

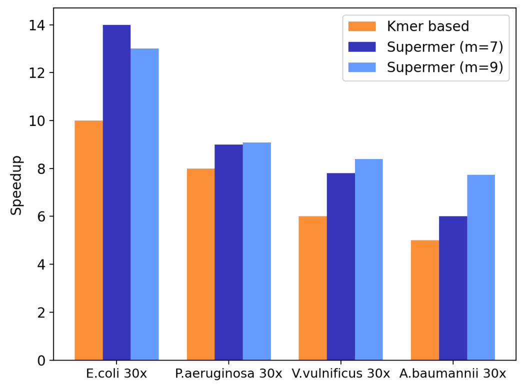
We use a variety of genomic input datasets, also used in previous work [6], [13] with $k = 17$, for our performance evaluation. The corresponding names and file sizes are shown in Table I.

Empirical results were collected on the IBM Power System AC922, Summit, at Oak Ridge National Laboratory. Each Summit node is a dual socket, with a 22-core IBM Power 9 CPU @3.07GHz, and 3 NVIDIA V100 GPUs per socket.

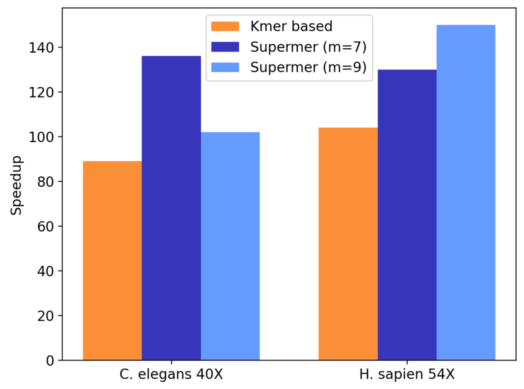
[2]https://github.com/PASSIONLab/DEDUKT

Short Name	Species and Strain	Fastq Size
E. coli 30X	*Escherichia coli* MG1655 strain	792 MB
P. aeruginosa 30X	*Pseudomonas aeruginosa* PAO1	360 MB
V. vulnificus 30X	*Vibrio vulnificus* YJ016	297 MB
A. baumannii 30X	*Acinetobacter baumannii*	249 MB
C. elegans 40X	*Caenorhabditis elegans* Bristol mutant strain	8.90 GB
H. sapien 54X	*Homo sapiens*	317 GB

TABLE I: Datasets used for performance evaluation.

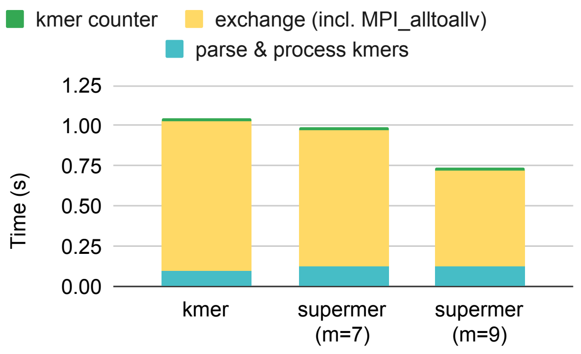


(a) 96 GPUs (6 per node) over CPU baseline utilizing 672 cores (42 per node)

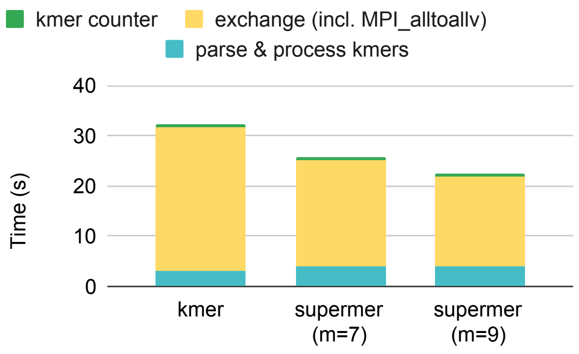


(b) 384 GPUs (6 per node) over CPU baseline utilizing 2,688 cores (42 per node).

Fig. 6: Speedup in overall performance (excl. I/O) over the CPU baseline k-mer counter



(a) *C. elegans* 40X



(b) *H. sapien* 54X

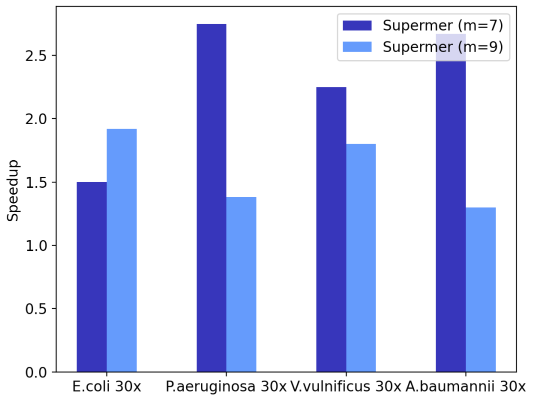
Fig. 7: Runtime breakdown of GPU based k-mer counters on 64 nodes with 384 GPUs.

Each node has 512 GB of DDR4 memory accessible to the CPUs. Each NVIDIA V100 GPU contains 80 streaming multiprocessors (SMs), 16 GB of high-bandwidth memory (HBM2), and a 6 MB L2 cache available to the SMs. On each node, CPUs are connected to the GPUs, and GPUs to GPUs, via NVIDIA's high-speed NVLink, providing a peak bandwidth of 25 GB/s per link. For the experiments with GPUs, each MPI rank executes on a single core and offloads its k-mer parsing, hashing, and counting to its GPU counterpart on the same socket. Data is transferred back from the GPU to its CPU core counterpart. The many-to-many k-mer and supermer exchanges are performed by CPU cores only. We use 6 MPI ranks per node (1 per GPU) in total for the experiments utilizing the GPUs. For the CPU baseline experiments, all 42 cores on node are utilized, with 1 MPI rank mapped to each core.

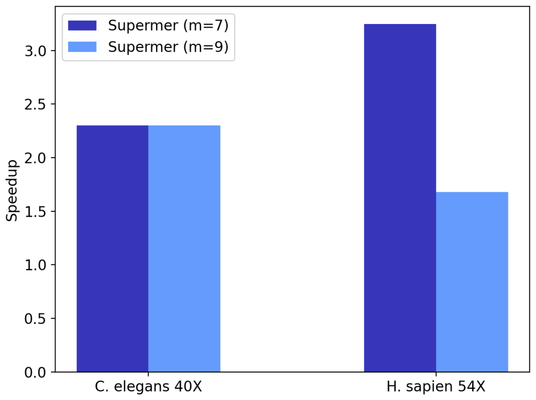
Nodes are connected via a Mellanox dual-rail EDR 100G Infiniband non-blocking fat-tree network, providing per node injection bandwidth of 23 GB/s. More details on the Summit architecture can be found on the Oak Ridge Leadership Computing Facility websites, https://www.olcf.ornl.gov and https://docs.olcf.ornl.gov. The codes were compiled with GCC-6.4 and IBM Spectrum MPI 10.3.1. NVCC 10.1 was used for compiling the CUDA code.

B. *Speedup in overall performance*

The distributed k-mer counters discussed in this paper primarily consist of three modules: (1) parsing k-mers (supermers), (2) copying data back and forth from CPU to GPU and perform MPI_Alltoall, and (3) populating the k-mer counter. Figure 6a compares the performance of these three modules excluding the I/O on 16 nodes of Summit cluster. The CPU baseline uses 672 cores in total (42 cores each node), and the speedups are shown on 96 GPUs (6 GPUs each) using k-mer-based and supermer-based k-mer counters. We observe $11\times$ and $13\times$ speedup on average using k-mer based, supermer based ($m = 7$ and $m = 9$) k-mer counter, respectively. Figure 6b compares the larger dataset of *H. sapien* 54X and *C. elegans* 40X on 64 nodes with 2688 CPU cores and 384 GPUs. For the *H. sapien* 54X dataset, our supermer based solution achieves up to $150\times$ speedup compare to the CPU based counterpart.



(a) 16 nodes with 6 MPI ranks per node (96 GPUs).



(b) 64 nodes with 6 MPI ranks per node (384 GPUs).

Fig. 8: Speedup of MPI_Alltoallv routine using supermers compared to k-mers

.

C. Comparison between GPU based k-mer counters

Figure 7b and 7a show the performance breakdown of the three main modules of the k-mer counter on *H. sapien* 54X and *C. elegans* 40X dataset, respectively. The figures compare runtime performance using k-mer-based and supermer-based k-mer counting on 64 nodes (384 GPUs). For both datasets of the supermer version, we observe a fair increase in the parse and process k-mer module (33% in *H. sapien* 54X) as building supermer requires extra steps including finding minimizers. A similar pattern is also observed in the k-mer counting phase where an extra parsing phase is required to extract k-mers from the received supermers, resulting in a 27% runtime increase for the human dataset. However, these costs are offset by the 33% speedup of the supermer-based data exchange module. Since this expensive phase accounts for up to 80% of the total time, results show an overall decrease in the run time using our approach. Detailed analysis of the communication reduction is discussed in the subsequent subsection.

	k-mer	supermer (m=9)	supermer (m=7)
E. coli 30X	412M	126M	108M
P. aeruginosa 30X	187M	56M	48M
V. vulnificus 30X	154M	47M	41M
A. baumannii 30X	129M	40M	34M
C. elegans 40X	4.7B	1.5B	1.3B
H. sapien 54X	167B	59B	50B

TABLE II: Total number of k-mers and supermers exchanged in the k-mer- and super-based counters

D. Reduction in communication volume and time

By using supermer-based k-mer counter, we can effectively reduce the communication volume. Table II presents the number of supermers using minimizer length of 9 and 7, as well as the number of k-mers from different datasets. Using a smaller minimizer length creates an opportunity to have longer but fewer supermers. Though this directly reduces the communication volume, it often increases work load imbalance. Note that this approach requires an extra byte of communication to identify the length of each supermer.

Results show a significant communication reduction of $4\times$ using a window length of 15 via the supermer approach. Figure 8a demonstrates the speedup from MPI_Alltoallv routine by using supermers with minimizer length of 9 and 7, compared to k-mers on 16 nodes with 96 GPUs on comparatively smaller datasets. Figure 8b presents the improvement over larger datasets on 64 nodes with 384 GPUs, highlighting up to a $3\times$ communication speedup for *H. sapien* 54X dataset. Note that the variance in the speedup is caused by the load imbalance of the k-mer distribution of the dataset.

E. Scalability on GPUs

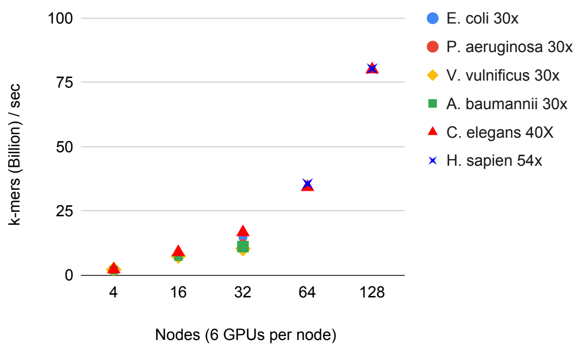


Fig. 9: Scalability of k-mer insertion rate (in Billions) using the computation kernels on GPUs (excl. exchange module) from 4 to 128 nodes. Both *C. elegans* 40X and *H. sapien* 54X achieve $2.3\times$ speedup on 128 nodes compared to 64 nodes.

Figure 9 presents the scalability of the GPU computation kernels ranging from 4 nodes (24 GPUs) to 128 nodes (768 GPUs). The smaller datasets (less than 1GB) use up to 32 nodes, and the larger sequences utilize up to 128 nodes. We observe linear speedup in almost all the datasets. For example, *C. elegans* 40X achieves $4\times$, $8\times$, $16\times$, $37\times$ speedup on 16,

	Average.	kmer		supermer (m=7)		Load
	#k-mers	min	max	min	max	Imbalance
C. elegans 40X	12M	12M	14M	3M	50M	1.16
H. sapien 54X	255M	253M	283M	41M	606M	2.37

TABLE III: Imbalance in the number of k-mers assigned to each partition (384 GPUs) using k-mer- and supermer-based counters. We computed the load imbalance as the ratio of the maximum load over the average load, where the load is defined as the number of k-mers.

32, 64 and 128 nodes, respectively. Both *C. elegans* 40X and *H. sapien* 54X datasets scale by 2.3$\times$ on 128 nodes from 64 nodes. However, a degradation in performance or non-linear speedup (*V. vulnificus* 30X scales 5$\times$ faster compare on 32 nodes to baseline 4 nodes, where *C. elegans* 40X scales 8$\times$ faster), are often caused by inherent skewness of the datasets which results in imbalanced partitions and higher k-mer counting overhead due skewed to k-mer profile.

Table III presents the minimum and maximum number of k-mers instances counted at each node. In the k-mer-based approach, the load imbalance in the number of k-mer counts across nodes in low (1.13) which is due to an even partitioning of data. However, using the supermer-based approach increases the load imbalance to 2.37. This is caused because of the skewness introduced in the partitioning due the use of minimizer-based partitioning. This will result in under utilization of nodes that receive a smaller portion of k-mers to count.

VI. Related Work

There are a plethora of approaches and implementations for counting k-mers on single-node shared-memory systems (see recent survey by Manekar and Sathe, 2018 [17]). Gerbil [8] and KMC3 [14] stand out as state-of-the-art tools that also employ GPUs and supermers, respectively, for k-mer counting. Further, they can accommodate datasets exceeding single node shared memory resources. In order to do so, however, each relies on file system resources, and incurs the egregious overheads thereof. Distributed memory k-mer counters, by contrast, support ever growing bioinformatics workloads by splitting data and computation across multiple compute nodes. Necessary communication is performed over network interconnects, designed for scalable parallel communication, unlike file systems.

Most k-mer counting solutions, for both shared and distributed memory, focus on counting k-mers from "second generation" sequences, and in particular, optimize for (and are limited to) their short lengths (typically 100-250 bps). "Third generation" sequence lengths by contrast are typically 1,000-100,000 bps, and can vary widely within the same input dataset. Our distributed memory k-mer counter is built using the histogram functionality of diBELLA's [7] k-mer analysis component, which supports both second and third generation sequence lengths.

Other noteworthy distributed memory k-mer counting tools include HipMer's k-mer analysis module [12] [11], Bloomfish [9], and Kmerind [21] [22]. HipMer's distributed memory k-mer analysis innovations, for second generation sequences, are incorporated and extended in diBELLA's k-mer analysis for third generation sequences. Both of these and this work are bulk-synchronous MPI. This work, however, focuses just on the k-mer counting aspect of the analyses, as do Bloomfish and Kmerind. Bloomfish is an alternative MapReduce-style distributed memory parallelization of the single-node k-mer counter, Jellyfish [19]. Kmerind is a hybrid MPI-OpenMP library that provides a number of optimized k-mer indexing capabilities for second generation sequences. None of these distributed memory tools employ supermers or GPU acceleration.

To the best of our knowledge, ours is the first work to accelerate k-mer counting with GPUs or to employ supermers in a distributed memory setting. Our analysis in Section V studies the scalability impacts thereof, particularly the affects on computation to communication ratios, in contrast to a CPU-only k-mer counter derived from diBELLA's k-mer analysis implementation.

VII. Conclusions

We presented a massively parallel algorithm for k-mer counting, a kind of string histogramming which is fundamental to many bioinformatic analyses. Our work combines distributed memory parallelization, GPU parallelism within a node, and a novel technique to improve spatial locality and reduce communication by hashing k-mers based on their context and communicating longer substrings called supermers. We believe this is the first use of either GPUs or supermers in the context of distributed memory parallelization for this problem. Our GPU optimizations successfully reduce overall run time by two orders of magnitude on the human genome data set, the largest example we use here. In general, the benefits of GPUs are strongest as the data sets grow, suggesting they will be increasingly important in future data sets involving several related species, microbial communities, or across a database of proteins or genomes.

Our GPU optimizations effectively turn a compute-bound problem into one dominated by communication. In particular, many-to-many k-mer exchange for redistributing k-mers tends to be the secondary bottleneck at small scales and the primary bottleneck at large scale of distributed memory k-mer counters [7], [10], [22], [33]. Our novel use of supermers in distributed memory parallelization is combined with GPU optimizations, improving communication costs by reducing communication volume. Prior approaches compute each k-mer before redistributing k-mers across processors, increasing the working data set size by an order of magnitude before communication. In contrast, we communicate larger supermer strings from which individual k-mers can be computed at the destination. We show a significant improvement of 1.5$\times$ using supermers on top of our GPU optimizations, resulting in overall speedups of up to 150$\times$.

Implemented as a general purpose k-mer counter, our tool can be used for counting k-mers in single genome, a microbial community (metagenome), comparisons to massive genome or protein databases, and the growing set of application problems and datasets involving sequence data.

In future work, we plan to investigate the issue of the high load imbalance introduced due to the use of supermers. We plan to devise a better partitioning algorithm that maintains the locality and at the same time partitions data evenly.

Acknowledgments

This work is supported by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE under contract number DE-AC02-05CH11231. We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DEAC02-05CH11231. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] F. Almodaresi, J. Khan, S. Madaminov, P. Pandey, M. Ferdman, R. Johnson, and R. Patro, "An incrementally updatable and scalable system for large-scale sequence search using lsm trees," *bioRxiv*, 2021.

[2] F. Almodaresi, P. Pandey, M. Ferdman, R. Johnson, and R. Patro, "An efficient, scalable, and exact representation of high-dimensional color information enabled using de bruijn graph search," *Journal of Computational Biology*, vol. 27, no. 4, pp. 485–499, 2020.

[3] G. Benoit, P. Peterlongo, M. Mariadassou, E. Drezen, S. Schbath, D. Lavenier, and C. Lemaitre, "Multiple comparative metagenomics using multiset k-mer counting," *PeerJ Computer Science*, vol. 2, p. e94, 2016.

[4] R. Chikhi, A. Limasset, and P. Medvedev, "Compacting de bruijn graphs from sequencing data quickly and in low memory," *Bioinformatics*, vol. 32, no. 12, pp. i201–i208, 2016.

[5] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, "Kmc 2: fast and resource-frugal k-mer counting," *Bioinformatics*, vol. 31, no. 10, pp. 1569–1576, 2015.

[6] M. Ellis, "Parallelizing irregular applications for distributed memory scalability: Case studies from genomics," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2020.

[7] M. Ellis, G. Guidi, A. Buluç, L. Oliker, and K. Yelick, "diBELLA: Distributed long read to long read alignment," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–11.

[8] M. Erbert, S. Rechner, and M. Müller-Hannemann, "Gerbil: a fast and memory-efficient k-mer counter with GPU-support," *Algorithms for Molecular Biology*, vol. 12, no. 1, p. 9, 2017.

[9] T. Gao, Y. Guo, Y. Wei, B. Wang, Y. Lu, P. Cicotti, P. Balaji, and M. Taufer, "Bloomfish: a highly scalable distributed k-mer counting framework," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2017, pp. 170–179.

[10] E. Georganas, "Scalable parallel algorithms for genome analysis," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2016.

[11] 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 *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–11.

[12] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de bruijn graph construction and traversal for de novo genome assembly," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 437–448.

[13] G. Guidi, M. Ellis, D. Rokhsar, K. Yelick, and A. Buluç, "Bella: Berkeley efficient long-read to long-read aligner and overlapper," *bioRxiv*, p. 464420, 2020.

[14] M. Kokot, M. Długosz, and S. Deorowicz, "Kmc 3: counting and manipulating k-mer statistics," *Bioinformatics*, vol. 33, no. 17, pp. 2759–2761, 2017.

[15] D. Li, C.-M. Liu, R. Luo, K. Sadakane, and T.-W. Lam, "MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph," *Bioinformatics*, vol. 31, no. 10, pp. 1674–1676, 2015.

[16] Y. Li *et al.*, "Mspkmercounter: a fast and memory efficient approach for k-mer counting," *arXiv preprint arXiv:1505.06550*, 2015.

[17] S. C. Manekar and S. R. Sathe, "A benchmark study of k-mer counting methods for high-throughput sequencing," *GigaScience*, vol. 7, no. 12, p. giy125, 2018.

[18] G. Marçais, D. DeBlasio, P. Pandey, and C. Kingsford, "Locality-sensitive hashing for the edit distance," *Bioinformatics*, vol. 35, no. 14, pp. i127–i135, 2019.

[19] G. Marçais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, vol. 27, no. 6, pp. 764–770, 2011.

[20] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in DNA sequences using a Bloom filter," *BMC bioinformatics*, vol. 12, no. 1, p. 333, 2011.

[21] T. Pan, P. Flick, C. Jain, Y. Liu, and S. Aluru, "Kmerind: A flexible parallel library for k-mer indexing of biological sequences on distributed memory systems," *IEEE/ACM transactions on computational biology and bioinformatics*, 2017.

[22] T. C. Pan, S. Misra, and S. Aluru, "Optimizing high performance distributed memory parallel hash tables for dna k-mer counting," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 135–147.

[23] P. Pandey, F. Almodaresi, M. A. Bender, M. Ferdman, R. Johnson, and R. Patro, "Mantis: A fast, small, and exact large-scale sequence-search index," *Cell systems*, vol. 7, no. 2, pp. 201–207, 2018.

[24] P. Pandey, M. A. Bender, R. Johnson, and Patro, "Squeakr: an exact and approximate k-mer counting system," *Bioinformatics*, vol. 34, no. 4, pp. 568–575, 2018.

[25] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "debgr: an efficient and near-exact representation of the weighted de bruijn graph," *Bioinformatics*, vol. 33, no. 14, pp. i133–i141, 2017.

[26] R. Patro, G. Duggal, M. I. Love, R. A. Irizarry, and C. Kingsford, "Salmon provides fast and bias-aware quantification of transcript expression," *Nature methods*, vol. 14, no. 4, pp. 417–419, 2017.

[27] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.

[28] M. Roberts, B. R. Hunt, J. A. Yorke, R. A. Bolanos, and A. L. Delcher, "A preprocessor for shotgun assembly of large genomes," *Journal of computational biology*, vol. 11, no. 4, pp. 734–752, 2004.

[29] B. Solomon and Kingsford, "Improved search of large transcriptomic sequencing databases using split sequence bloom trees," in *International Conference on Research in Computational Molecular Biology*. Springer, 2017, pp. 257–271.

[30] B. Solomon and C. Kingsford, "Fast search of thousands of short-read sequencing experiments," *Nature Biotechnology*, vol. 34, no. 3, p. 300, 2016.

[31] C. Sun, R. S. Harris, R. Chikhi, and P. Medvedev, "Allsome sequence bloom trees," *Journal of Computational Biology*, vol. 25, no. 5, pp. 467–479, 2018.

[32] D. E. Wood and S. L. Salzberg, "Kraken: ultrafast metagenomic sequence classification using exact alignments," *Genome biology*, vol. 15, no. 3, pp. 1–12, 2014.

[33] K. Yelick, A. Buluç, M. Awan, A. Azad, B. Brock, R. Egan, S. Ekanayake, M. Ellis, E. Georganas, G. Guidi *et al.*, "The parallelism motifs of genomic data analysis," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2166, p. 20190394, 2020.