Bloomfish: A Highly Scalable Distributed K-mer Counting Framework

Tao Gao,[a,d] Yanfei Guo,[b] Yanjie Wei,[f] Bingqiang Wang,[e] Yutong Lu,[d,e,g]
Pietro Cicotti,[c] Pavan Balaji,[b] and Michela Taufer[a]

[a]University of Delaware
[b]Argonne National Laboratory
[c]San Diego Supercomputer Center
[d]National University of Defense Technology
[e]National Supercomputing Center in Guangzhou
[f]Shenzhen Institutes of Advanced Technology, CAS
[g]Sun Yat-sen University

***Abstract*—K-mer counting is a fundamental operation in DNA research and genome analytics; its application includes estimating genome assembly, understanding similarities in genomic samples, and merging a newly processed genome with a reference genome. As the genome dataset becomes larger and larger, designing a highly optimized distributed-memory implementation becomes more and more important. Current distributed-memory solutions have two limitations: they have a high memory footprint, and they do not provide advanced optimizations for loading enormous genome datasets into memory. Based on these observations, we present Bloomfish, a distributed, memory-efficient, scalable solution to the limits of current work. To keep a low memory footprint, Bloomfish leverages the compact hash array design of the single-node Jellyfish system and the optimized workflow of the high-performance MapReduce framework Mimir. We have also codesigned Mimir's I/O to efficiently load enormous datasets. We ran Bloomfish on the Tianhe-2 supercomputer with large sequence datasets (up to 24 TB). Our results show that Bloomfish achieves unprecedented scalability in genome analytics.**

Keywords: K-mer counting; Genome analysis; MapReduce; Memory efficiency; I/O optimization; Performance and scalability

I. Introduction

In traditional linguistics, when dealing with strings, the term k-mer refers to the set of possible substrings of length k in the string. In genomics, a string is a DNA sequence built from adenine, guanine, cytosine, and thymine; and k-mer refers to the set of possible subsequences built from the four nucleobases with length k. K-mer counting is a fundamental operation in genome analytics, with several use cases. It can be used to analyze and estimate genome assembly, such as the de novo genome assembler [1]. K-mer counting also is a core tool for understanding similarities in genomics samples [2] (e.g., rate of increase in k-mer counts explains how similar multiple genomes are). Moreover, k-mer counting is an error validation tool [3]; for example, when a newly processed genome is merged with a reference genome, a drastic increase in k-mer counts indicates the presence of errors in the sequence.

Computationally the total size of intermediate data (i.e., the set of k-mers) can be significantly larger than the input DNA sequences, negatively impacting the memory footprint of k-mer counting executions. Single-node solutions rely on the development of compact k-mer storage methods [4], the reduction in number of stored k-mers by filtering out singleton k-mers [5], and the development of disk-based algorithms to store the intermediate data in permanent storage [6], [7], [8], [9], [10], [11]. State-of-art highly optimized single-node codes such as Jellyfish [12] combine these techniques, resulting in an efficient memory footprint for small datasets; but they cannot efficiently count k-mers for larger and larger datasets. Figure 1 shows the k-mer counting performance of Jellyfish on a single node of the Tianhe-2 supercomputer. The figure shows that Jellyfish takes about one day to count k-mers for a modest 3 TB dataset from the 1000 Genomes project [13].

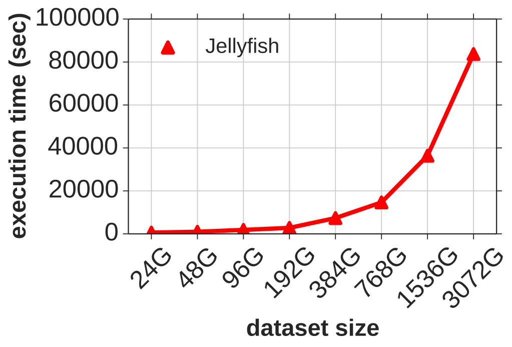


Fig. 1: Performance of a 22-mer counting for a 3 TB dataset with Jellyfish on a single node of the Tianhe-2 supercomputer.

Distributed-memory solutions overcome the size limit of single-node codes by using larger aggregated memories on clusters or supercomputers [14], [15], [16], [17]. Often these solutions adopt methods traditionally used in single-node implementations in order to keep a low memory footprint. For example, the Hipmer package [15], [16] includes a k-mer counting module that eliminates singleton k-mers by a Bloom filter, but it does not use compact k-mer store methods such as the one used in Jellyfish and does not optimize the workflow to minimize the intermediate data staging. A more recently released k-mer code, Kmerind [17], relies on multiple intermediary stages and a high memory footprint, limiting the data that can be processed per node (e.g., a 3 GB input dataset on a 128 GB node of the XSEDE cluster Comet). To the best of our knowledge, none of the existing distributed-memory solutions adaptively deals with I/O variability associated with the different response times of different processes to read data in parallel file systems. As a result, the processing times of entire datasets are bound to the processing time of the slowest process.

In this paper we tackle these problems by developing a new distributed-memory k-mer counting framework called Bloomfish. Bloomfish is a memory-efficient, scalable k-mer counting framework that leverages the highly compact intermediate data storage solution of Jellyfish and the optimized workflow of a MapReduce framework Mimir. Moreover, to support loading terabytes and petabytes of genomic data into memory efficiently, we redesigned the I/O framework of Mimir. We note that these I/O optimizations can be applied to MapReduce applications other than k-mer counter applications. Experimental evaluation proves that Bloomfish significantly scales to enormous datasets (up to 24 TB) compared with both single-node and distributed-memory implementations; it also substantially reduces the memory footprint compared with that of other distributed-memory implementations.

The contributions of this paper are as follows.

1) We present a highly scalable and memory-efficient k-mer counting framework, called Bloomfish.
2) We codesign the I/O framework on which Bloomfish is built to support a stream I/O model, work stealing for adaptive I/O performance, and I/O overlapping with communication and computation.
3) We compare Bloomfish with state-of-the-art single-node and distributed-memory implementations.
4) We present Bloomfish's weak and strong scalability on two high-end platforms, the XSEDE supercomputer Comet and the supercomputer Tianhe-2, for DNA sequence datasets up to 24 TBs—larger than any previously tested.

II. Background

In this section, we provide a high-level overview of the Jellyfish framework, MapReduce programming model, and Mimir implementation of MapReduce.

A. Jellyfish Overview

Jellyfish [4] [12] is a multithreading k-mer counting framework. It achieves much lower memory usage by designing a lock-free and compact k-mer counting hash table. In the hash table, the ith possible storage position for a given mer m is

$$pos(m,i) = (hash(m) + reprobe(i))\, mod\, M.$$

In this equation, M is the length of the hash table, and $reprobe$ function is used to solve hash collision. For each hash entry, Jellyfish uses a bit-packeted data structure to reduce the memory usage. In addition, because most k-mers appear a few times, Jellyfish uses a small count field and allows one mer to have more than one entry in the hash table. For example, if one mer has two entries, $\langle mer, c_1\rangle$ and $\langle mer, c_2\rangle$, then the count for this mer is c_1c_2.

Jellyfish also uses a space-efficient method to encode the mers. In Jellyfish, the length of the hash array is always a power of 2. Suppose $M = 2^l$. Each k-mer is encoded as an integer in this set, $U_k = [0, 4^k - 1]$. Jellyfish chooses a hash function, $hash(m) = f(m)\, mod\, M$, in which f is a bijection function (i.e., f: $U_k \Rightarrow U_k$). Since $f(m)$ is a bijection, the mer m can be computed by $f(m)$. Hence, given the value $reprobe(i)$, the position of a $\langle mer, count\rangle$ pair in the hash array encodes the lower l bits of $f(m)$. Jellyfish stores only the $2k - l$ higher bits of $f(m)$ and the reprobe count $i+1$ (0 is reserved to indicate empty) for the mer. The mer is recovered by computation when necessary. With this design, Jellyfish can considerably reduce the amount of memory required to store the mers.

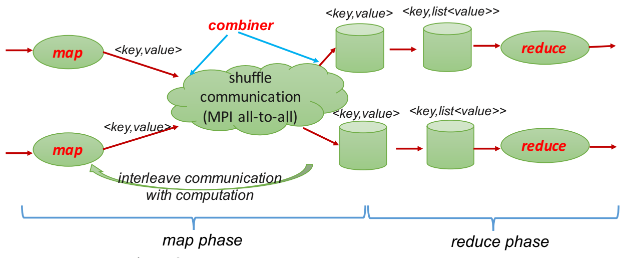
B. MapReduce and Mimir Overview

MapReduce is a programming model intended for data-intensive applications [18]. It has proven suitable for a wide variety of applications because of its simplicity, scalability, and fault tolerance. A MapReduce job usually involves three phases: *map*, *shuffle*, and *reduce*. The *map* phase processes the input data using a user-defined map callback function and generates intermediate $\langle key, value\rangle$ pairs. The *shuffle* phase performs an all-to-all communication that distributes the intermediate $\langle key, value\rangle$ pairs across all processes. In this phase, $\langle key, value\rangle$ pairs with the same key are also merged and stored in $\langle key, list\langle value\rangle\rangle$ lists. The *reduce* phase processes the lists with a user-defined reduce callback function and generates the final output. The user needs to implement the map and reduce callback functions, while the MapReduce runtime handles the parallel job execution, communication, and data movement.

Successful implementations of the MapReduce model include Hadoop [19] and Spark [20]. However, these frameworks are designed for cloud computing systems. Different from these implementations, Mimir [21] is a MapReduce implementation targeted for supercomputing systems. It is built on top of MPI [22], which utilizes core high-performance computing (HPC) principles to optimize data movement. Moreover, it is developed in the C++ language, which makes interoperability with existing HPC applications, such as Jellyfish, easier.

In Mimir, MapReduce jobs are divided into two phases: `map` and `reduce`. The `shuffle` communication is included in the `map` phase, while the conversion of $\langle key, value\rangle$ pairs to $\langle key, list\langle value\rangle\rangle$ lists is included in the `reduce` phase. The workflow of Mimir is shown in Figure 2. In the `map` phase, Mimir uses a pipeline design to interleave the shuffle communication with computation in the `map` callback. The intermediate $\langle key, value\rangle$ pairs are stored in an intermediate database. In the `reduce` phase, the intermediate $\langle key, value\rangle$ pairs are converted to $\langle key, list\langle value\rangle\rangle$ lists first. Then the reduce callback is applied to each such list. Mimir provides two optional optimizations: combiner optimization (called compression and partial reduction in [21]) and KV-hint optimization. The combiner optimization merges intermediate $\langle key, value\rangle$ pairs with the same key before and after the shuffle communication. The KV-hint optimization uses the type information of key and value to reduce the storage and communication size of the intermediate data.

Our previous work on Mimir focused on optimizing the memory usage of the framework. In that implementation, we adopted the discrete I/O model, in which different files are partitioned to different processes statically. This I/O model is suitable for datasets with many equal-sized files. At the same time,


combiner
<key,value>
<key,list<value>>
map
reduce
shuffle communication (MPI all-to-all)
interleave communication with computation
map phase
reduce phase
Fig. 2: MapReduce workflow in Mimir.

we use collective communication (i.e., MPI_Alltoallv) to perform the shuffle communication of intermediate $\langle key, value \rangle$ pairs.

III. Design of Bloomfish

As discussed in Section I, two major challenges arise in designing efficient k-mer counting frameworks for distributed-memory systems. The first challenge is to use memory efficiently. To solve this challenge, Bloomfish leverages two optimizations of existing tools. Specifically, Bloomfish reuses the highly compact intermediate data storage in Jellyfish [4] and builds on top of the memory-efficient MapReduce implementation Mimir [21]. The intermediate data storage in Jellyfish can significantly reduce the memory requirement to store the k-mers, and the highly pipelined workflow in Mimir can significantly reduce the intermediate data staging. With this design, Bloomfish can use memory much more efficiently than existing distributed-memory k-mer counting tools can. Another challenge is to load enormous datasets from disk to memory efficiently. We design the stream I/O model, work stealing, and I/O overlapping with communication and computation to solve this problem. The optimizations are implemented in the MapReduce framework Mimir. Note that other applications built on top of Mimir also can use these optimizations.

A. Bloomfish Overview

Bloomfish is a port of Jellyfish over the Mimir MapReduce framework. It inherits most of the local k-mer management and statistical analysis framework from Jellyfish and uses Mimir for performing I/O and data movement between processes. It achieves a low memory footprint by reusing the k-mer storage method designed by Jellyfish and using the pipeline design in Mimir.

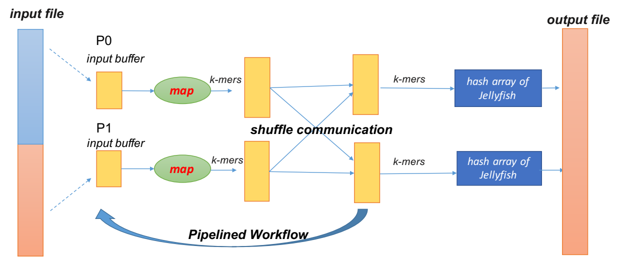


Fig. 3: Workflow of Bloomfish.

The workflow of Bloomfish is shown in Figure 3. In the workflow, the input files are read and passed to the `map` function. The intermediate mers are generated in the `map` callback. These mers are shuffled over the network, and the same mers are sent to the same processes. Once received, these mers are inserted into the compact hash array, and the counting is implemented. After the counting is finished, the results are stored in permanent storage. Note that the k-mer counting is implemented in a pipelined way. Bloomfish can read partial data and perform the disk I/O, communication, and computation in a pipelined way. Thus, all the other buffers except the intermediate data hash array are independent of the intermediate data size. In this way, Bloomfish minimizes the intermediate data staging compared with the existing distributed-memory work.

Because most k-mers usually appear a few times (e.g., the average appearance time of each 22-mer is just 16 for the 3 TB 1000 Genomes dataset), the combiner optimization (which performs local counting before communication) cannot reduce the shuffle communication size. As a result, the combiner optimization is not applied to the workflow. To allow the integration of the Jellyfish hash array into Mimir, we extend Mimir to support customized intermediate data containers. Specifically, applications can provide a customized object with `read` and `write` interfaces as intermediate storage.

Like Jellyfish, Bloomfish supports two DNA file formats: fasta and fastq [23]. Each sequence in the fasta file contains a header line and one or multiple sequence lines. The header line always starts with a “>” character. In the fastq format, each sequence contains four lines: a header line, which starts with an “@” character; a sequence line; a separator line, which starts with a “+” character; and a quality score line.

B. I/O Performance Improvement

In the preceding section, we discussed how Mimir uses memory efficiently by reusing Jellyfish and Mimir. In this section, we discuss how to solve the I/O bottleneck problem. We note that the optimizations are done in Mimir. Thus, the approach is general to other applications based on Mimir as well.

Some problems with performance loss are due to disk I/O in HPC platforms with parallel file systems. One problem is associated with the file size of sequence datasets, which can vary greatly. For example, the maximum file size in the HG00097 sequence dataset from the 1000 Genomes project is about 80 GB, whereas the minimum file size is about 100 MB. Thus, the discrete I/O model (which partitions the different files to different processes) may result in some processes partitioning much more data than do other processes. To solve this problem, we propose the stream I/O model (which views files as if they are segments of a continuous stream of data).

Another problem is the performance variability of I/O operations in the parallel file system. Figure 4 illustrates this by using 768 processes to read 6 TB input data (each process reads about 8 GB) at the same time. As shown in the figure, the read time varies greatly even if the input data size for each process is the same. Performance variability is a common issue of the parallel file system [24] and results in some processes slowing the execution of the whole program. We design a work-stealing method to solve this problem. If one process finishes the work, it will try to steal work from other processes. Our work-stealing method is based on the stream I/O model and implemented with MPI-3 one-sided communication interfaces. The performance variability of I/O operations also results in high global synchronization overhead. We use a nonblocking communication mechanism to overlap I/O with communication. The nonblocking communication is implemented with the MPI nonblocking collective communication interfaces `MPI_Ialltoall` and `MPI_Ialltoallv`.

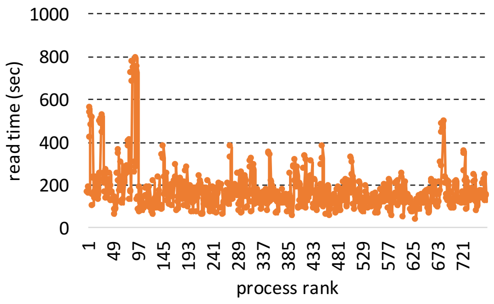


Fig. 4: Time to read 6 TB input data with 768 processes (8 GB per process) on the Tianhe-2 supercomputer.

1) Stream I/O Model Support: Instead of partitioning different files to different processes, as is done in the discrete I/O model, in the stream I/O model we partition various chunks to different processes, with each process getting the same amount of data. One issue of the stream I/O model is that one input record (e.g., a line in the text files) may be split into two chunks. We propose a repartition method to handle this situation. Applications just need to set a `repartition` callback to decide the number of bytes sent to the previous chunk. Then, Mimir implements repartition automatically. The implementation of repartition is determined by the scheduling methods. In Section III-B2, we describe the repartition implementation in the work-stealing algorithm.

The `repartition` callback is designed for two sequence file formats: fasta and fastq. In the fasta file format, if the first character of next line is ">," meaning that the cutting point is in the last line of one sequence, the repartition method sends bytes until the end of the current line; otherwise, it sends out the current line plus $k-1$ bytes in the next line. In the fastq format, each sequence contains only four lines. The repartition function sends bytes until the start of the next sequence. Since the quality score line may start with "@," deciding the start point of the next sequence is tricky. However, we can use the features of the fastq file format: the first character of the line following the header line is not "@," but the first character of the line following the quality score line is "@." Thus, when we get a line started with "@" in the fastq format, we check the first character of the following line to decide whether the current line is a quality score line or a header line.

The stream I/O model allows us to implement different scheduling methods. One of the simplest scheduling methods is to partition the same number of continuous chunks to each process statically. In this scheduling method, the repartition bytes are always sent to the previous process $p-1$, in which p is the rank of the current process.

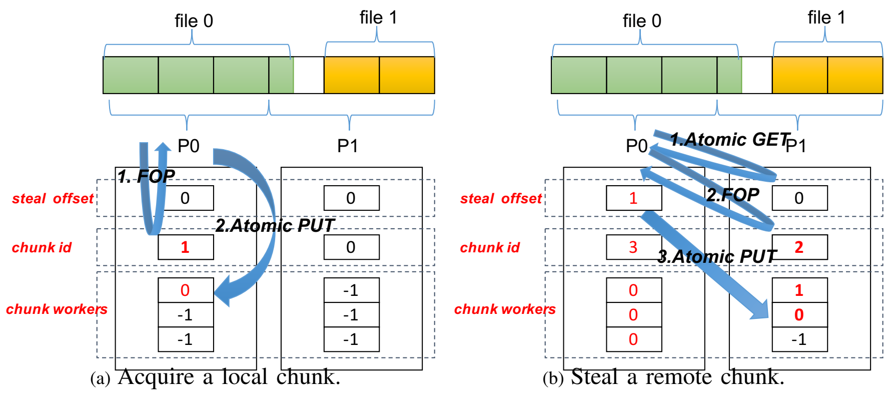
2) Work-Stealing Support: To address processing time skew caused by the I/O performance variability, we design a work-stealing algorithm based on the stream I/O model. The file chunks are partitioned to different processes statically, and each process gets the same number of continuous chunks. The process that owns the chunks is called the owner process of the chunks. The chunks statically partitioned to a process are called local chunks of that process. We identify the chunks partitioned to other processes as remote chunks of that process. Each chunk is represented by a two-dimensional structure `(owner rank, chunk id)`. The `chunk id` is a local index of the chunk in the owner process. All processes have the metainformation of the chunks. Given the `owner rank` and the `chunk id`, the process can get the file name and offset of that chunk easily. Each process works on its local chunks first. If one process finishes processing all the chunks statically partitioned to it, then it will steal remote chunks from other processes.

Each process has three data structures in the work-stealing algorithm: `steal offset`, `chunk id`, and `chunk workers`. The `steal offset` symbolizes the rank offset of the victim process relative to the current process. The `chunk id` represents the local index of next chunk in this process. The `chunk workers` are an array in which the ith item stores the process rank to get the ith chunk (either the owner process or a stealer process). A process that gets a chunk is called the worker process of that chunk. We reserve -1 to represent the worker process that is still unknown.

The work-stealing algorithm is shown in Figure 5. In this figure, two files (represented by different colors) with six chunks are taken as an example. These six chunks are partitioned to two processes, and each process gets three chunks. Figure 5a shows process 0 acquiring a local chunk. Figure 5b shows process 0 stealing a remote chunk from process 1.

As shown in Figure 5a, when acquiring a local chunk, the process uses `FOP` (i.e., `fetch_and_op`) to add and fetch the `chunk id` atomically. If the fetched `chunk id` is less than the number of chunks partitioned to the process, then this process gets this chunk successfully. Once acquiring the chunk, the process atomically `PUT`s the process rank into the corresponding item of `chunk workers`.

Otherwise, if the acquiring of the chunk fails, then the process tries to steal chunks from other processes, as shown in Figure 5b. The stealer process `s` will start from a `steal offset` of 1 to n-1 (in which n is the number of processes). The stealer process chooses the process $(steal\ offset + s)\ mod\ n$ as the victim process v. For example, in Figure 5b, when the `steal offset` is 1, process 0 chooses $(1+0)\ mod\ 2 = 1$ as the victim process. During the stealing, the stealer process `s` atomically `GET`s the `steal offset` of the victim process first. This is used to check whether there are unprocessed chunks on that process. If the `steal offset` of the victim process is zero, the victim process has not begun stealing. In other words, some unprocessed chunks probably are in the victim process. Thus, the stealer process will use `FOP` to fetch and add the `chunk id` of the victim



(a) Acquire a local chunk. (b) Steal a remote chunk.

Fig. 5: Design of work stealing.

process. If the returned value is less than the total number of chunks partitioned to the victim process, the stealing succeeds. Then, the stealer process atomically `PUT`s the rank into the corresponding item of the `chunk workers` in the victim process. If the `steal offset` of the victim process is not zero, processes in this range $[v, (v + steal\ offset)\ mod\ n)$ have finished processing their local chunks. As a result, the stealer process will move to process $(v + steal offset) mod n)$ by adding the `steal offset` of the victim process to its `steal offset`. This method is used to avoid duplicate steals from the processes without unprocessed chunks.

The repartition implementation in the work-stealing algorithm is based on the `chunk workers` structure. First, the chunk is loaded into memory. If the process is not the first chunk of a file and the previous chunk is not processed by the same process, then the process uses atomic `GET` to get the previous chunk's worker process from the `chunk workers`. Next, the process sends the repartition bytes to that worker process. Note that the previous chunk's worker process may still not be known. In this situation, Mimir will store the repartition bytes and try to send the repartition bytes the following time when a chunk is required. If a chunk is not the last chunk of a file, the process will receive repartition bytes before finishing the processing of the chunk. Two situations are possible. In the first situation the next chunk belongs to the same process and has still not been processed yet. Then the process will use `CAS` (i.e., `compare_and_swap`) to acquire the next chunk atomically. If this succeeds, then it will merge the remaining bytes in the current chunk with the next chunk directly, and no repartition is needed. In the second situation the next chunk will be processed by other processes. In this situation, the process uses atomic `GET` to get the worker rank of the next chunk from the `chunk workers` and waits to receive the repartition bytes from that process. Note that the process will need to wait only a short time because the worker process will send out the repartition bytes once the chunk is loaded into memory.

We implement the work-stealing algorithm with MPI 3 one-sided communication interfaces. The `FOP` and `CAS` are implemented with `MPI_Fetch_and_op` and `MPI_Compare_and_swap`. The `MPI_Accumulate` and `MPI_Get_accumulate` are used as atomic `PUT` and `GET`.

3) I/O and Communication Overlap: In the previous design of Mimir, we used blocking collective communication (i.e., `MPI_Alltoallv`) to exchange intermediate $\langle key, value\rangle$ data. However, the global synchronization of blocking collective communication introduces high overhead due to the I/O performance variability. For example, if a process is blocked by one I/O operation and other processes are performing the shuffle communication, then all the other processes need to wait for the completion of the blocking I/O operation before they can continue.

To address this performance issue, we designed a nonblocking collective shuffle communication mechanism. In the blocking collective implementation, each shuffle communication is composed of two MPI functions: `MPI_Alltoall` and `MPI_Alltoallv`. `MPI_Alltoall` is used to communicate send and receive counts used by the `MPI_Alltoallv`; `MPI_Alltoallv` is used to exchange the intermediate $\langle key, value\rangle$ data. In the nonblocking implementation, we use the nonblocking version of these two functions (i.e., `MPI_Ialltoall` and `MPI_Ialltoallv`) to perform the shuffle communication.

A variable N buffer method is used to overlap I/O with communication. Each process starts with two communication buffers. When the first buffer is full, the process starts a shuffle communication and starts working on the second buffer. When the second buffer is full, the process checks whether the first shuffle communication is complete. If it is complete, then the process reuses the first buffer. If the first shuffle communication is not complete, the process allocates a new (third) buffer and starts working on it. In this way, each time the current buffer is full, the process allocates a new communication buffer if none of the previous shuffle communications has been completed. To avoid some processes allocating too many communication buffers, Mimir sets a maximum number of buffers each process is allowed to allocate. This design means that all processes might not have the same number of communication buffers, and it also prevents very symmetric applications from allocating too many buffers.

After issuing the `MPI_Ialltoall` function, Mimir tests its status periodically. If `MPI_Ialltoall` finishes and all the `MPI_Ialltoallv` functions of the earlier shuffle communications have been issued, Mimir invokes the corresponding MPI_Ialltoallv of this shuffle communication. Note that MPI_Ialltoallv must be invoked in the same order as the corresponding MPI_Ialltoall. After invoking MPI_Ialltoallv, Mimir checks its status periodically. If MPI_Ialltoallv of a shuffle communication has finished, then the corresponding communication buffers can be reused. Note that different processes may invoke MPI_Ialltoall and MPI_Ialltoallv in different order. As a result, MPI_Ialltoall and MPI_Ialltoallv are operated on two different duplicate communicators to address the order restriction of collective communication on the same communicator in the MPI standard.

By using the nonblocking collective communication methods, we overlap the I/O with shuffle communication.

IV. Comparison

In this section, we compare Bloomfish with the single-node implementation Jellyfish [4] and the distributed implementation Kmerind [17]. Jellyfish is chosen as the state-of-art single-node implementation because of its low memory footprint and high performance. Kmerind [17] is chosen because it is a general k-mer count framework and has been proven better than other works, such as Kmernator [14]. We use the real sequence dataset from the 1000 Genomes project [13] for our tests.

A. Platforms, Datasets, and Settings

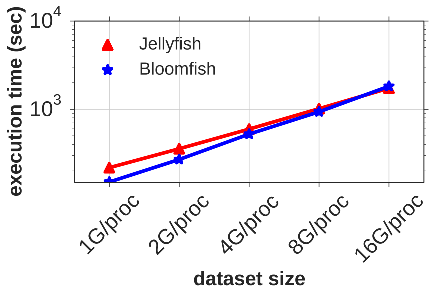
Our experiments were performed on the XSEDE cluster Comet [25] and on the supercomputer Tianhe-2. Comet is an NSF Track2 system located at the San Diego Supercomputer Center. Each compute node has two Intel Xeon E5-2680v3 CPUs (12 cores each, 24 cores total) running at 2.5 GHz. Each node has 128 GB of memory and 320 GB of flash SSDs. The nodes are connected with Mellanox FDR InfiniBand, and the parallel file system is Lustre. Tianhe-2 is a high-performance supercomputer located at the National Supercomputer Center in Guangzhou. Each compute node has two Intel Xeon E2-2692v2 CPUs (12 cores each, 24 cores total) running at 2.2 GHz. Each node has 64 GB of memory. The nodes are connected with Tianhe express-2 [26], and the parallel file system is H2FS [27]. We use MPICH 3.2 [28] for the tests on Comet and MPICH 3.1.3 with a customized GLEX channel on the Tianhe-2 [29].

For the Bloomfish configuration, the chunk size and communication buffer size are set to 64 MB for all tests, and the maximum number of communication buffers is set to 5. The k is set to 22, the same as in the Jellyfish paper [4]. For all tests, we run one thread on one core for Jellyfish and one process on one core for Kmerind [17] and Bloomfish. We set the hash array size of Bloomfish to 512 million for all these tests.

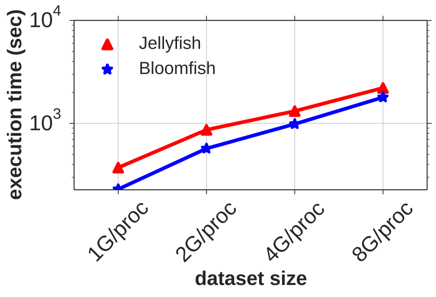
B. Single-Node Comparison with Jellyfish

In this section, we compare the execution time when running Jellyfish with 24 threads and Bloomfish with 24 processes. The single-node execution times of the two codes are shown in Figure 6 for Comet and Tianhe-2. Note that the y-axis is in logarithm scale. As shown in the figure, both Jellyfish and Bloomfish achieve linear scalability. On Comet, Bloomfish achieves better performance than Jellyfish does for datasets less than 96 GB (4 GB/proc) and shows similar performance for larger datasets up to 384 GB (16 GB/proc). The reason is that the file sizes in the 1000 Genomes dataset vary greatly. Jellyfish assigns different files to different threads. As a result, some threads process much more data than do other threads. However, Bloomfish adopts a better I/O framework than Jellyfish does. Bloomfish not only partitions data evenly when the file sizes vary greatly but also supports work stealing to reduce the impact of processing time skew caused by I/O performance variability.

On Tianhe-2, Bloomfish outperforms Jellyfish for all datasets from 1 GB/proc to 8 GB/proc. Note that Tianhe-2 nodes have up to 64 GB memory and thus we tested only up to 8 G/proc.



(a) Comet



(b) Tianhe-2

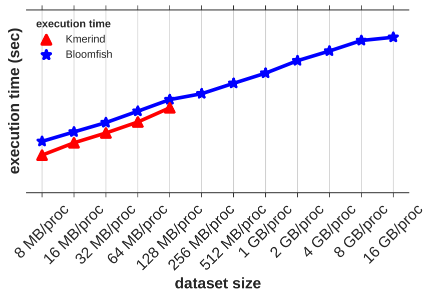
Fig. 6: Single-node results on Comet and Tianhe-2 (24 threads for Jellyfish and 24 processes for Bloomfish) using the 1000 Genomes dataset.

These tests prove that Bloomfish can achieve performance that is better than or comparable to that of the highly optimized single-node implementation for datasets up to 384 GB. Results presented in Figure 1 (in the introduction of this paper), on the other hand, show how Jellyfish performance quickly deteriorates on single nodes for large datasets. Specifically, the figure points out how the k-mer counting of up to 3 TB on Tianhe-2 can take up to one day when using Jellyfish, making the need for a faster, distributed implementation vital for the increasingly large DNA sequences. A similar pattern can be observed for Comet.

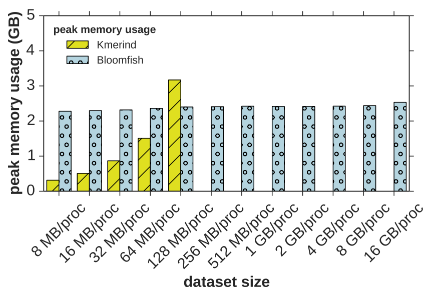
C. Single-Node and Distributed-Memory Comparisons with Kmerind

In this section, we compare Bloomfish with Kmerind [17] on a single node and on multiple nodes of Comet. Note that we can use at most 64 nodes (1,536 processes) on Comet. We use only Comet for this comparison because Kmerind currently cannot compile successfully on Tianhe-2. The MPI I/O method of Kmerind gains the best performance based on our tests. As a result, we set Kmerind to use MPI I/O for the following tests.

Figure 7 shows the single-node execution time and peak memory usage for the two codes. As shown in the figure,



(a) Execution Time



(b) Peak Memory Usage per Process

Fig. 7: Single-node results on Comet (24 processes for Kmerind and Bloomfish).

for datasets up to 3 GB the execution time of Bloomfish is comparable to or slightly slower than that of Kmerind. The reason is that the I/O performance is not a problem for small datasets. Bloomfish can process up to 384 GB datasets (16 GB/process) on a single node, but Kmerind can process only up to 3 GB datasets (128 MB/process). The reason Bloomfish can process datasets 128X times larger than those of Kmerind is illustrated in Figure 7b, which plots the peak memory usage of both codes. As shown in the figure, the memory usage of Bloomfish is kept constant. The reason is that we set the hash array size large enough to hold the largest dataset (i.e., 384 GB) and the other memory usage (e.g., input and communication buffers) is independent of intermediate data sizes. However, Kmerind's peak memory usage increases rapidly. This increase is because Kmerind does not use highly compact k-mer storage method and has extra intermediate data stating in the workflow. In other words, Bloomfish uses the highly optimized intermediate data storage solution and optimizes the workflow to minimize the intermediate data staging.

Since Kmerind can process only much smaller (i.e. 128X times) datasets per node compared with Bloomfish, we use its distributed-memory implementation and the maximum dataset Kmerind can handle with 1,536 processes (i.e., 192 GB) to perform its distributed-memory comparison with Bloomfish. To measure the resource efficiency of the two codes, we use as the metric of success the aggregated CPU-hours, namely, (number of cores) * (execution time). The results are shown in Figure 8. As we can see, Kmerind can process the 192 GB input only with 1,536 processes. However, Bloomfish can process the same dataset with from 1,536 to only 24 processes. The aggregated CPU-hours of Bloomfish for all settings are better than those for Kmerind. Specifically, Kmerind need about 32.0 CPU-hours to finish the k-mer counting of 192 GB with 1,536 processes, whereas Bloomfish needs only 17.6 CPU-hours with 1,536 processes and 8.0 CPU-hours with 24 processes. This improvement is results because Bloomfish uses memory much more efficiently and designs highly optimized I/O methods.

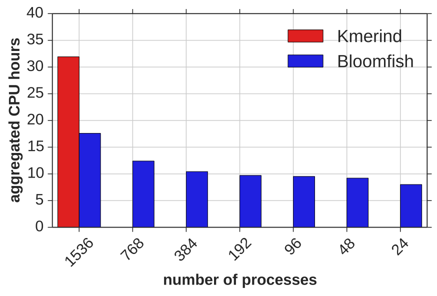


Fig. 8: Aggregated CPU-hours for the k-mer counting of 192 GB DNA sequences with Kmerind and Bloomfish.

In summary, our tests with the 1000 Genomes dataset prove that Bloomfish's execution times are comparable to those of Kmerind for the single-node tests when using small datasets. Unlike Kmerind, Bloomfish can process larger datasets on a single node. The Bloomfish's distributed-memory implementation uses resources (i.e., memory and CPU) much more efficiently than Kmerind does.

V. Scalability

To evaluate Bloomfish's strong scalability and weak scalability, we ran tests using unprecedentedly large DNA sequences.

A. Platforms, Datasets, and Settings

Our scalability tests were performed on Comet and Tianhe-2 (see Section IV). Because we wanted to study the scalability of increasingly large datasets, we used two datasets: the real sequence dataset from the 1000 Genomes project [13] described in Section IV and a new synthetic dataset generated by the ART simulator [30], [31] that can scale up to terabytes. For the synthetic data, we simulated sequence data generated by Illumina machines [32] and used a human genome data set [33] as the reference genome. The Mimir configuration was the same as in Section IV.

B. Strong Scalability and Speedup

In this section, we present the strong-scalability results on Comet and Tianhe-2. A DNA sequence dataset of one individual (i.e., HG00096) was chosen for the strong-scalability tests. This data set contains 15 files and has a total size of about 64 GB. The execution times from 24 processes to 768 processes are presented. The speedup of one setting was calculated by dividing the execution time with 24 processes by the execution time with that setting.

Strong-scalability results on Comet are shown in Figure 9a. We can see that Bloomfish's speedup increases up to 192 processes and then slows from 384 processes. The reason is that the I/O variability has a larger impact as the execution time becomes shorter and shorter. The execution time with 768 processes is about 90 seconds. The same strong-scalability tests were performed on Tianhe-2; the results are shown in Figure 9b. Here we see that Bloomfish achieves much better speedup on Tianhe-2. The speedup is close to ideal up to 384 processes: the speedup with 384 processes compared with 24 processes is 12.6x. With 768 processes, however, the I/O variability has a larger impact on the execution time, and the speedup slows. The execution time with 768 processes is approximately 43 seconds. During the tests, we observed worse I/O performance on Comet than on Tianhe-2, which explains the worse speedup on Comet.

In summary, Bloomfish achieves almost ideal speedup up to 384 processes on Tianhe-2. The I/O performance, however, limits the strong scalability when the execution time becomes short.

C. Weak Scalability and Efficiency

We next describe weak-scalability test results on Tianhe-2. We used two settings (i.e., 1 GB/process and 8 GB/process) for the two kinds of datasets. For the 8 GB/process tests of the 1000 Genomes dataset, we duplicated the 3 TB real dataset 10 times to get a 30 TB dataset. The efficiency of one setting was calculated by dividing the execution time with 24 processes by the execution time with that setting.

The results of the 1 GB/process setting are shown in Figure 10. We can see that Bloomfish scales to 3,072 processes for both the 1000 Genomes dataset and the synthetic dataset. As shown in Figure 10a, the efficiency of the 1000 Genomes dataset up to 384 processes is close to 100%, and the efficiency for some settings is even larger. The reason is that the mer distribution in the 1000 Genomes dataset is not uniform. Thus, when the dataset size doubles, the computation and communication do not double in some situations. For example, we observe the least shuffle communication times with the 192 processes setting. The efficiency drops from 768 processes because the I/O performance becomes worse; however, the efficiency still is about 41% for 3,072 processes. For the 3 TB dataset, Bloomfish counts 22-mers in about 14 minutes with 3,072 cores. In contrast, Jellyfish needs about 1 day with 24 cores.

We performed similar tests with the synthetic dataset; the results are shown in Figure 10b. We observe a pattern similar to that for the 1000 Genomes dataset: the efficiency is about 43% with 3,072 processes.

We also perform weak-scalability tests by increasing the number of datasets per process. The results with 8 GB/process are shown in Figure 11. The 1000 Genomes dataset is generated by duplicating the real dataset 10 times. The results are shown in Figure 11a. The efficiency is close to 100% until 384 processes, and some settings have an efficiency larger than 100%. The reason is that the mer distribution in the dataset is not uniform, as explained before. The efficiency begins dropping with 768 processes, due mainly to the I/O performance; but the efficiency is still about 49% with 3,072 processes.

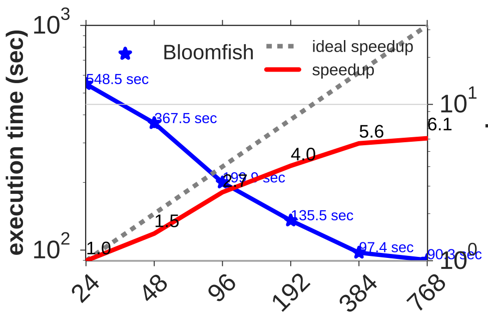
We performed similar tests with the synthetic dataset; the results are shown in Figure 11b. The pattern is similar, and the efficiency is 56% with 3,072 processes. We note that Bloomfish counts 22-mers of the 24 TB dataset with 3,072 processes in just one hour.

In summary, Bloomfish enables scientists to study unprecedentedly large DNA datasets with HPC systems. Specifically, we prove that Bloomfish can scale to at least 24 TB datasets and 3,072 processes.

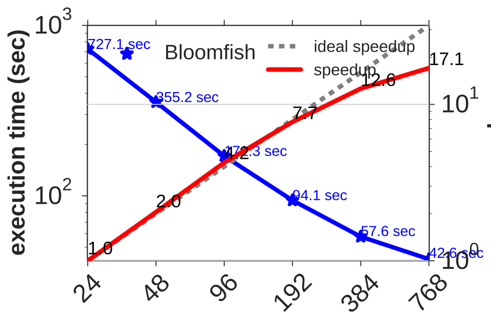
VI. Related Work

Numerous efforts have focused on k-mer counting on single-node systems. These efforts can be divided into three categories: (1) optimizing the intermediate k-mer storage by designing compact data structure [4]; (2) using some probabilistic method to filter out part of k-mers and reduce the number of k-mers that need be stored [5], [34]; and (3) using disk as intermediate data storage [6], [7], [8], [9], [10], [11]. Different from these single-node works, we aim to design efficient k-mer counting for high-performance distributed-memory systems. We reuse the compact hash array design of a highly optimized single-node implementation (i.e., Jellyfish). We have proven that we can scale to distributed-memory systems without performance loss for single-node processing.

Other efforts aim to perform k-mer counting on distributed-memory systems. FASTdoop [35] extends Hadoop to process fast and fastq files with the MapReduce programming model. BioPig [36] and SeqPig [37] process sequencing data as the Pig queries on top of Hadoop. SparkSW [38] implements the Smith-Waterman (SW) algorithm on Spark for parallel processing of sequencing data. However, these approaches all rely on MapReduce-derived frameworks, which are not generally available on supercomputers. Kmernator [14] is a MPI toolkit for large-scale genomic analysis that uses MPI to perform k-mer counting on distributed-memory systems. Hipmer [15], [16] is a de novo genome assembler that supports k-mer counting as a substage of the workflow. It also introduces some optimizations to the distributed-memory implementation, such as eliminating erroneous k-mers with the Bloom filter, parallel reading input files with MPI I/O, and processing high-frequency k-mers separately. However, some of the optimizations are specific to the genome assembly problem. For example, error validation does not support eliminating singleton k-mers. Kmerind [17] is a distributed-memory k-mer indexing tool, and the method it uses to build a count index can be viewed as k-mer counting. Kmerind optimizes the problem with a distributed hash table and various I/O methods. While these implementations provide baseline performance for distributed-memory k-mer counting, they are not highly optimized for keeping a low memory footprint and

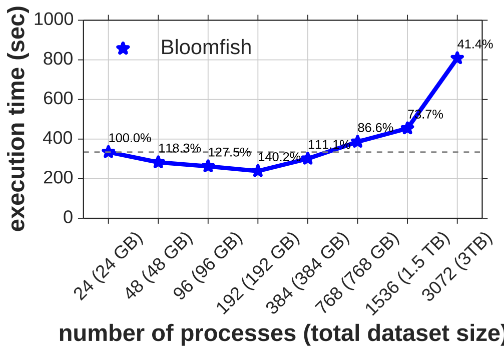


(a) Comet

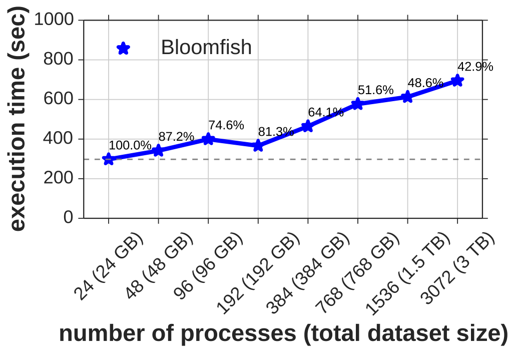


(b) Tianhe-2

Fig. 9: Strong scalability on Comet and Tianhe-2.

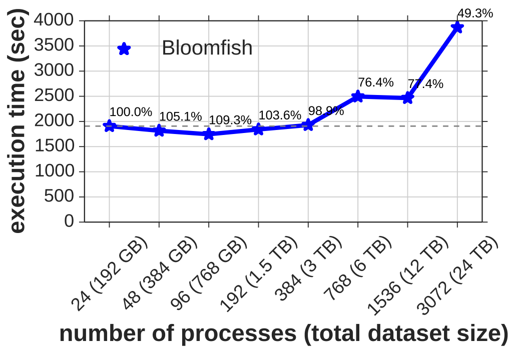


(a) 1000 Genomes dataset

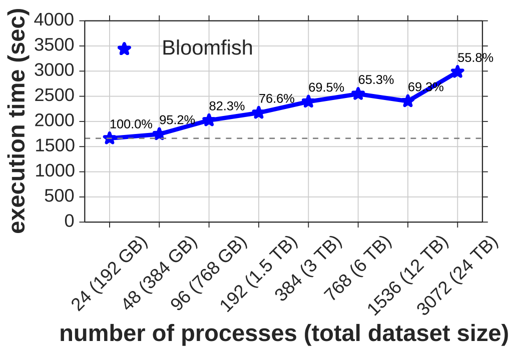


(b) Synthetic dataset

Fig. 10: Weak scalability (1 GB/process).



(a) 1000 Genomes dataset (duplicate 3 TB dataset)



(b) Synthetic dataset

Fig. 11: Weak scalability (8 GB/process).

loading enormous data into memory. Some optimizations used to reduce memory usage are specific to typical problems, such as filtering out singleton k-mers. Instead, Bloomfish keeps a low memory footprint by leveraging the optimized hash array design to reduce the intermediate data size and by using a pipelined workflow to reduce intermediate data staging.

Other works [16], [17] use parallel I/O to load large input data into memory. However, the method they use cannot adapt to I/O performance variability. Instead, we redesign the I/O framework in Mimir to gain high I/O performance even when the I/O performance variability is severe. Moreover, the I/O optimization is general to other applications built on top of Mimir. With these optimizations, Bloomfish can process up to 24 TB input data in about one hour.

VII. Conclusion

We design a highly memory-efficient and scalable k-mer counting framework, Bloomfish, on top of a high-performance MapReduce framework, Mimir. Bloomfish leverages the compact k-mer storage solution of a highly optimized single-node implementation Jellyfish and the optimized workflow of Mimir to keep a low memory footprint. We also codesign Mimir's I/O to load enormous datasets from parallel file systems to memory more efficiently. Our results show that Bloomfish achieves unprecedented scalability: it can count 22-mers of a 24 TB dataset with 3,072 processes in about one hour. To the best of our knowledge, this is the largest dataset ever reported for k-mer counting problems. Since k-mer counting is the primary substage of various genome analysis applications, this work has the potential to contribute to genomic analysis at the terabyte and petabyte scale.

Acknowledgment

Yanfei Guo and Pavan Balaji were supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. Pietro Cicotti, Tao Gao, and Michela Taufer were supported by NSF grants #1318445 and #1318417. Tao Gao was also supported by China Scholarship Council. Yutong Lu was supported by National Key R&D Project in China 2016YFB1000302. Yanjie Wei was supported by Science Technology and Innovation Committee of Shenzhen under grant No. JCYJ20160331190123578 and 2016B090918122. Part of the research in this paper used resources of the National Supercomputer Center in Guangzhou, China. XSEDE resources, supported by NSF grant ACI-1053575, were used to obtain some other performance data. We thank Zhiguang Chen, Peihao Liu, and Peijun Yang from the National Supercomputer Center in Guangzhou for their help in the tests on the Tianhe-2 supercomputer; Jintao Meng from Shenzhen Advanced Technology Research Institute for guiding the selection of the simulator; Rob Latham from Argonne National Laboratory for sharing his optimization experience in improving I/O performance; and Gail Pieper from Argonne National Laboratory for her editorial feedback.

References

[1] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen *et al.*, "De novo assembly of human genomes with massively parallel short read sequencing," *Genome Research*, vol. 20, no. 2, pp. 265–272, 2010.

[2] H.-J. Yu, "Segmented k-mer and its application on similarity analysis of mitochondrial genome sequences," *Gene*, vol. 518, no. 2, pp. 419–424, 2013.

[3] Y. Liu, J. Schröder, and B. Schmidt, "Musket: A multistage k-mer spectrum-based error corrector for Illumina sequence data," *Bioinformatics*, vol. 29, no. 3, pp. 308–315, 2013.

[4] 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.

[5] 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.

[6] S. Kurtz, A. Narechania, J. C. Stein, and D. Ware, "A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes," *BMC Genomics*, vol. 9, no. 1, p. 517, 2008.

[7] G. Rizk, D. Lavenier, and R. Chikhi, "DSK: K-mer counting with very low memory usage," *Bioinformatics*, vol. 29, no. 5, p. 652, 2013.

[8] S. Deorowicz, A. Debudaj-Grabysz, and S. Grabowski, "Disk-based k-mer counting on a PC," *BMC Bioinformatics*, vol. 14, no. 1, p. 160, 2013.

[9] Y. Li and X. Yan, "MSPKmercounter: a fast and memory efficient approach for k-mer counting," *arXiv preprint arXiv:1505.06550*, 2015.

[10] P. Audano and F. Vannberg, "KAnalyze: A fast versatile pipelined k-mer toolkit," *Bioinformatics*, vol. 30, no. 14, pp. 2070–2072, 2014.

[11] 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.

[12] "Jellyfish Website," http://www.cbcb.umd.edu/software/jellyfish/.

[13] "1000 Genomes project," http://www.internationalgenome.org.

[14] "Kmernator," https://github.com/JGI-Bioinformatics/Kmernator.

[15] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 437–448.

[16] 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 *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. IEEE, 2015, pp. 1–11.

[17] 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," in *Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*. ACM, 2016, pp. 422–433.

[18] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[19] "Apache Hadoop," http://hadoop.apache.org/.

[20] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[21] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "Mimir: Memory-efficient and scalable MapReduce for large supercomputing systems," in *Proceedings of the 31th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[22] "MPI: A message-passing interface standard," http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[23] "DNA sequence format," https://www.genomatix.de/online_help/help/sequence_formats.html.

[24] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *IEEE 28th International on Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 155–164.

[25] "Comet Cluster," http://www.sdsc.edu/support/user_guides/comet.html.

[26] X.-K. Liao, Z.-B. Pang, K.-F. Wang, Y.-T. Lu, M. Xie, J. Xia, D.-Z. Dong, and G. Suo, "High performance interconnect network for Tianhe system," *Journal of Computer Science and Technology*, vol. 30, no. 2, p. 259, 2015.

[27] W. Xu, Y. Lu, Q. Li, E. Zhou, Z. Song, Y. Dong, W. Zhang, D. Wei, X. Zhang, H. Chen, J. Xing, and Y. Yuan, "Hybrid hierarchy storage system in milkyway-2 supercomputer," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 367–377, 2014.

[28] "MPICH Library," http://www.mpich.org.

[29] M. Xie, Y. Lu, K. Wang, L. Liu, H. Cao, and X. Yang, "Tianhe-1a interconnect and message-passing services," *IEEE Micro*, vol. 32, no. 1, pp. 8–20, 2012.

[30] W. Huang, L. Li, J. R. Myers, and G. T. Marth, "ART: A next-generation sequencing read simulator," *Bioinformatics*, vol. 28, no. 4, pp. 593–594, 2012.

[31] "ART simulator," https://www.niehs.nih.gov/research/resources/software/biostatistics/art/.

[32] "Illumina," https://www.illumina.com.

[33] "Reference genome dataset," ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/reference//human_g1k_v37.fasta.gz.

[34] Q. Zhang, J. Pell, R. Canino-Koning, A. C. Howe, and C. T. Brown, "These are not the k-mers you are looking for: Efficient online k-mer counting using a probabilistic data structure," *PloS One*, vol. 9, no. 7, p. e101271, 2014.

[35] U. Ferraro Petrillo, G. Roscigno, G. Cattaneo, and R. Giancarlo, "FASTdoop: A versatile and efficient library for the input of FASTA and FASTQ files for MapReduce Hadoop bioinformatics applications," *Bioinformatics*, vol. 33, no. 10, pp. 1575–1577, 2017.

[36] H. Nordberg, K. Bhatia, K. Wang, and Z. Wang, "Biopig: a hadoop-based analytic toolkit for large-scale sequence data," *Bioinformatics*, vol. 29, no. 23, pp. 3014–3019, 2013.

[37] A. Schumacher, L. Pireddu, M. Niemenmaa, A. Kallio, E. Korpelainen, G. Zanetti, and K. Heljanko, "SeqPig: Simple and scalable scripting for large sequencing data sets in Hadoop," *Bioinformatics*, vol. 30, no. 1, pp. 119–120, 2014.

[38] G. Zhao, C. Ling, and D. Sun, "SparkSW: Scalable distributed computing system for large-scale biological sequence qlignment," in *ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE/ACM, May 2015, pp. 845–852.