# DISTRIBUTED MEMORY BUILDING BLOCKS FOR MASSIVE BIOLOGICAL SEQUENCE ANALYSIS

A Dissertation
Presented to
The Academic Faculty

By

Tony C. Pan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Computational Science and Engineering

Georgia Institute of Technology

May 2018

**DISTRIBUTED MEMORY BUILDING BLOCKS FOR MASSIVE BIOLOGICAL SEQUENCE ANALYSIS**

Approved by:

Dr. Srinivas Aluru, Advisor
School of Computational Science and Engineering
*Georgia Institute of Technology*

Dr. David Bader
School of Computational Science and Engineering
*Georgia Institute of Technology*

Dr. Ümit Çatalyürek
School of Computational Science and Engineering
*Georgia Institute of Technology*

Dr. King Jordan
School of Biology
*Georgia Institute of Technology*

Dr. Fredrick Vannberg
School of Biology
*Georgia Institute of Technology*

Dr. Richard Vuduc
School of Computational Science and Engineering
*Georgia Institute of Technology*

Date Approved: March 27, 2018

People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.

*Donald Knuth*

For Katy, Colin, and Alex.

## ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude and love for my wife, Katy, and my children, Colin and Alex. Their love, support, and sacrifice gave me the courage to undertake and perseverance to complete this journey.

I am equally grateful to my advisor, Dr. Srinivas Aluru for giving me the opportunity to realize my doctoral pursuit at a critical point of my professional career. Dr. Aluru's encouragement, support, and guidance have been exemplary, and instrumental in the success of this work and my growth as a researcher.

I would like to thank Dr. David Bader, Dr. Ümit Çatalyürek, Dr. King Jordan, Dr. Fredrick Vannberg, and Dr. Richard Vuduc for their advices, patience, and time. Their knowledge and experience have helped shape this research and my perspective of the bioinformatics and high performance computing fields.

I would also like to thank Dr. Sanchit Misra from Intel Parallel Computing Lab for his expertise and assistance throughout our collaboration to optimizing the Kmerind library. I am grateful to my fellow graduate students and post-doctoral fellows and research scientists in the Aluru group for the open discussions and friendship. In particular, I would like to thank Patrick Flick, Rahul Nihalani, Chirag Jain, and Yongchao Liu for the many technical discussions and collaborations.

Finally, I would like to thank my parents and parents-in-law, my extended family, and my friends for their support, emotional and otherwise, for Katy, Colin, Alex, and me.

This research is supported in part by the National Science Foundation under IIS-1416259, CNS-1229081, CCF-1361053, and Intel Parallel Computing Center grant on Big Data in Biosciences and Public Health.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

## LIST OF ALGORITHMS

## SUMMARY

$K$-mer indices and de Bruijn graphs are important data structures in bioinformatics with multiple applications ranging from foundational tasks such as error correction, alignment, and genome assembly, to knowledge discovery tasks including repeat detection and SNP identification. While advances in next generation sequencing technologies have dramatically reduced the cost and improved latency and throughput, few bioinformatics tools can efficiently process the data sets at the current generation rate of 1.8 terabases every 3 days.

The volume and velocity with which sequencing data is generated necessitate efficient algorithms and implementation of $k$-mer indices and de Bruijn graphs, two central components in bioinformatic applications. Existing applications that utilize $k$-mer counting and de Bruijn graphs, however, tend to provide embedded, specialized implementations. The research presented here represents efforts toward the creation of the first reusable, flexible, and extensible distributed memory parallel libraries for $k$-mer indexing and de Bruijn graphs. These libraries are intended for simplifying the development of bioinformatics applications for distributed memory environments. For each library, our goals are to create a set of API that are simple to use, and provide optimized implementations based on efficient parallel algorithms. We designed algorithms that minimize communication volume and latency, and developed implementations with better cache utilization and SIMD vectorization.

We developed Kmerind, a $k$-mer counting and indexing library based on distributed memory hash table and distributed sorted arrays, that provide efficient `insert`, `find`, `count`, and `erase` operations. For de Bruijn graphs, we developed Bruno by leveraging Kmerind functionalities to support parallel de Bruijn graph construction, chain compaction, error removal, and graph traversal and element query.

Our performance evaluations showed that Kmerind is scalable and high performance. Kmerind counted $k$-mers in a 120GB data set in less than 13 seconds on 1024 cores, and in-

dexing the $k$-mer positions in $17$ seconds. Using the Cori supercomputer and incorporating architecture aware optimizations as well as MPI-OpenMP hybrid computation and overlapped communication, Kmerind was able to count a $350$GB data set in $4.1$ seconds using $4096$ cores. Kmerind has been shown to out-perform the state-of-the-art $k$-mer counting tools at $32$ to $64$ cores on a shared memory system.

The Bruno library is built on Kmerind and implements efficient algorithms for construction, compaction, and error removal. It is capable of constructing, compacting,and generating unitigs for a $694$ GB human read data set in $7.3$ seconds on $7680$ Edison cores. It is $1.4\times$ and $3.7\times$ faster than its state-of-the-art alternatives in shared and distributed memory environments, respectively. Error removal in a graph constructed from an $162$ GB data set completed in $13.1$ and $3.91$ seconds with frequency filter of $2$ and $4$ respectively on $16$ nodes, totaling $512$ cores.

While our target domain is bioinformatics, we approached algorithm design and implementation with the aim for broader applicabilities in computer science and other application domains. As a result, our chain compaction and cycle detection algorithms can feasibly be applied to general graphs, and our distributed and sequential cache friendly hash tables as well as vectorized hash functions are generic and application neutral.

# CHAPTER 1
# INTRODUCTION AND MOTIVATION

The wide-spread adoption of next-generation sequencing (NGS) technologies in diverse disciplines, ranging from basic biology and medicine to agriculture and even social sciences, has resulted in the tremendous growth of public and private genomic data collections such as the Cancer Genome Atlas [1], the 1000 Genome Project [2], and the 10K Genome Project [3], and even clinical research and health care [4]. The ubiquity of NGS technology adoption is attributable to increases in sequencing throughput and decreases in cost. For example, one Illumina HiSeq X Ten system can sequence over 18,000 human genomes in a single year at less than $\$1000$ per genome, corresponding to approximately 1.6 quadrillion DNA base pairs per year.

The volume and the velocity at which genomes are sequenced continues to push bioinformatics as a *big data* discipline. Efficient management and timely processing of biological sequence data using sophisticated bioinformatic algorithms and tools that support data-driven computational tasks [5, 6] are essential for high throughput and low latency analyses. Some foundational bioinformatics tasks include whole genome assembly [7, 8, 9], sequencing coverage estimation and error correction [10, 11, 12], variant detection [11] and metagenomic analysis [13].

These challenges necessitate sophisticated algorithms and well implemented tools that can scale to ever increasing NGS sequence data sizes. This research focuses on two specific components that are central to many biological sequence analysis: $k$-mer indexing and de Bruijn graph construction and traversal. We begin with a review of related works.

## 1.1 Related Works

### 1.1.1 K-mer Indexing

Central to many bioinformatic tasks are $k$-mer (defined as a length $k$ sequence) counting and indexing, which is widely used in data processing tasks such as sequence alignment [14, 15], NGS read error correction [11, 12, 16], NGS read alignment [17, 18, 19], and *de novo* genome assembly [20, 7, 8, 9]. $K$-mer counting and indexing have found utility in other applications such as sequencing coverage estimation [10] and single nucleotide variant identification [11]).

$K$-mer Counting and indexing has been extensively investigated in the literature due to its centrality in many bioinformatic algorithms. Most algorithms and software for $k$-mer analysis target shared memory systems, and operate serially [10, 21, 22, 23, 24, 25] or use multiple threads [26, 27, 28, 29, 30, 31, 32, 33]. To the best of our knowledge, Kmernator [34] is currently the only stand-alone distributed memory $k$-mer counter available.

These software target different use cases and provide different interfaces and functionalities. While $k$-mer indexing software [10, 22, 23] can be used for $k$-mer counting, the converse may not hold true. A $k$-mer analysis software may also be designed for a specific pipeline, e.g. assembly, and therefore provides only application-specific query interfaces [31, 34], or only supports off-line queries [26].

In our review, we excluded tools that estimate abundance histograms only and do not support count or position queries, such as KmerGenie [35] and KmerStream [36].

*K-mer Counting*

Jellyfish [26] is a popular in-memory $k$-mer counter and introduces a lock-free hash table to support thread-level concurrent updates. It reduces memory consumption by using a bijective hash function that allows the lower bits of a key to be reconstructed from its hash bucket identifier, thus only the upper bits need to be stored. The memory usage is further

minimized by widening the data type only when the $k$-mer frequency is high. KCMBT [25] is an in-memory $k$-mer counter that employs cache efficient burst tries.

Out-of-core approach for reducing memory footprint includes the use of disks as external memory. Tools in this group operate with separate partitioning and counting phases. KAnalyze [28] counts $k$-mers in each input file block and stores the intermediate results, which are aggregated during the counting phase. DSK [27], MSPKmerCounter [24], KMC 2 [31], KMC 3 [33], and Gerbil [32] assign $k$-mers to on-disk buckets during partitioning, and process each bucket individually during the counting phase. Counting is accomplished in a majority of the tools mainly through incremental updates to hash tables, while some adopted sorting and aggregation [33].

Probabilistic data structures such as Bloom filters and Count-min Sketch [37] have also been used to reduce memory requirements. BFCounter [21] and Turtle [30] are two examples using Bloom filters. JellyFish 2 [26] also provides an option to support this technique. As Bloom filters can introduce false positives, a second scan of the $k$-mer counts is necessary. Khmer [29] is a $k$-mer counter based on Count-min Sketch that, while memory efficient, can overestimate $k$-mer counts.

A majority of the prior efforts focus on minimizing memory usage or efficient utilization of multiple CPU cores, or both. Memory limitations may be addressed via succinct data structures [26, 21, 30, 29] such as Bloom filters [38] and Count-min Sketch [37], via disk-based algorithms [27, 31, 33, 32], or via distributed memory algorithms [39]. Effective utilization of available cores may be achieved via thread-safe updates [26, 21], or via data partitioning followed by independent sequential computation. Partitioning may occur on disk [27, 31, 33, 32], or in memory [31, 33, 30, 39].

*K-mer Position Indexing*

Besides $k$-mer counts, some works further trace the position of each $k$-mer occurrence through string indexing data structures. Tallymer [10] and Gk-Array [22] are two tools

based on enhanced suffix array [40] (an equivalent representation of suffix tree). Välimäki and Rivals [23] extended Gk-Array by proposing a compressed representation based on FM-index [41] that further improves memory efficiency.

Suffix trees and arrays are not well-suited for distributed-memory $k$-mer counting and indexing. Distributed-memory suffix array construction has been demonstrated previously [42]. However, distributed query processing requires $O(\log(n))$ iterations of sequence comparison, each iteration requiring communication with remote processors. Kmerind instead stores $k$-mers in data structures that support local associative look-up or comparison-based searches using $k$-mers as keys.

*Distributed $K$-mer Counting and Indexing*

The $k$-mer counting and indexing tools discussed in sections 1.1.1 target shared-memory systems. Kmernator [34] is a hybrid MPI+OpenMP application that implements node- and thread-level master-slave work assignment. However, Kmernator only supports FASTQ files and canonical $k$-mers.

Distributed-memory assemblers often embed $k$-mer indexing and counting capability for erroneous $k$-mer removal and de Brujin graph construction. Examples of such assemblers include ABySS [8], PASHA [43] and HipMer [9]. ABySS uses hash tables, PASHA adopts a combination of hash tables and sorted vectors, while HipMer associates hash tables with Bloom filters. As these $k$-mer indexing and counting procedures are specialized for assembly only, none of them provides general purpose indexing APIs.

### 1.1.2 De Bruijn Graph

Initially suggested during the era of Sanger sequencing [44, 45], de Bruijn graph based *de novo* genome assembly approaches was popularized by Velvet [7] and have become the mainstay of analyzing high-throughput short reads from next generation DNA sequencing instruments. *De novo* genome assemblers based on de Bruijn graphs are more memory

and computational efficient when compared to their overlap-layout-consensus graph based alternatives.

Briefly, *de novo* genome assembly aims to reconstruct the true genomic sequence from sequence fragments, called *reads*, produced by a genome sequencer. In the overlap-layout-consensus approach, each read is aligned to every other read to construct a graph where vertices are reads and edges indicate suffix-prefix overlaps, and the assembled genome is obtained by finding a Hamiltonian path in the graph. In contrast, each vertex in a de Bruijn graph represents a distinct $k$-mers in the read data set, and each edge represent a $(k-1)$ overlap between two nodes. Assembly is theoretically obtained as a Euler tour of the graph. Practically, the assembled sequences are obtained by first finding linear paths and then by mapping reads to the linear paths to disambiguate between paths at branching points.

Most of de Brujin graph based assemblers employed this representation [46], such as Velvet, ABySS [8], SOAPdenovo [47, 48], IDBA [49] and Meraculous [50]. In contrast, some assemblers such as SPAdes [51, 52], ALLPATHS-LG [53] and MaSuRCA [54] adopted another representation by considering distinct $k$-mers as edges and $(k-1)$-mers as nodes. In this representation, each edge connects the two nodes that exactly match its $(k-1)$-length prefix and $(k-1)$-length suffix, respectively. Sohn and Nam [46] revealed that the latter approach tends to have fewer branching nodes than the former for large genomes. For example, by constructing a de Bruijn graph from the human reference genome, their example test with $k = 45$ showed that the former can have as much as two folds more branches than the latter, as the former approach with $k = 45$ is conceptually equivalent to the latter approach with $k = 46$. Regardless, both representations require expensive computation and large memory space.

In terms of computational time, Gnerre *et al.* [53] reported that for human whole-genome assembly, the multi-threaded ALLPATHS-LG took about 3 weeks wall-clock times with 48 processors, in comparison with about 3 wall-clock days taken by the multi-threaded SOAPdenovo [47] on the same hardware. As for memory consumption, existing assemblers

targeting large genomes typically work well with $< 512$ GB shared memory for human genome assembly [48, 55]. To improve speed, one commonly used approach is parallelization via multi-threading in shared-memory computers or distributed computing on clusters. To reduce memory overhead, recent efforts focus on using succinct data structures to realize memory-efficient de Brujin graph representations (e.g. [56, 50, 48, 57, 58, 59]). While memory efficient, these assemblers still suffer from long execution times [59], arguably exacerbated by the overhead associated with the data structures.

*Distributed Memory Genome Assemblers*

A few parallel and distributed de novo assemblers have been developed based on de Brujin graphs. ABySS [8] is the first work implemented using message passing interface (MPI), which constructs a distributed-memory de Brujin graph structure by compacting all edges per node into 8 bits with one bit indicating the presence or absence of one of the eight edges in two directions. However, ABySS builds edges by checking all possible neighbors per node, even if two $k$-mers are not adjacent in any input read. In this way, spurious edges might be introduced. In addition, ABySS does not batch the communication during traversal, thus suffering from high communication latency.

Ray [60] is a parallel and distributed de novo assembler with peer-to-peer communication at the core and implemented on a custom distributed computing framework.

PASHA [43] uses a similarly distributed de Brujin graph representation to ABySS, but builds edges only from the adjacency information of $k$-mers in the input reads. This assembler takes advantage of hybrid computing architectures consisting of shared-memory multi-CPUs and distributed-memory clusters to overcome memory and speed constraints.

YAGA [61] constructs a distributed de Brujin graph represented as a collection of edges. This algorithm implements a parallel list ranking approach to generate *unitigs* via path walking.

SWAP-Assembler [62] introduced a multi-step bi-directed graph (MSG) and a small

world asynchronous parallel (SWAP) computational framework to realize strong scaling up to one thousand cores for human genome assembly. SWAP uses a lock-compute-unlock mechanism to avoid conflicts when one edge is accessed by two simultaneous merge operations initiated by two processes.

HipMer [63, 9] is a parallelized version of Meraculous [50] and implemented using MPI and Unified Parallel C (UPC) [64] for various assembly tasks. HipMer merges partial unitigs from two processors by assigning both partial unitigs to one of the two processor arbitrarily. The freed processor can then initiate unitig extension with a randomly selected vertex. Initial unitig generation can also produce an *oracle_hash* function, which can later be used to map $k$-mers from the same unitig to the same processor, thus increasing data locality and reduce communication volume. HipMer has been shown to scale to tens of thousands of cores for human and wheat genome assembly.

Except for YAGA and SWAP-Assembler, the other parallel assemblers all provide end-to-end de novo assembly by enabling *contig* scaffolding with the help of paired-end information. Unlike other end-to-end assemblers that perform scaffolding using only paired-end reads, HipMer relies on a reference genome to facilitate the determination of location and orientation of *unitigs* on scaffolds.

*Graph Compaction and Unitig Generation*

De Bruijn graph based assemblers generally incorporate similar steps: including graph construction, error removal, chain compaction and unitig generation, gap closing and unitig generation, and extension through scaffolding using paired end information. The steps are often formulated as a pipeline and which may even exist as a script.

Chain compaction identifies linear chains in a de Bruijn graph that can be traversed unambiguously during assembly. A compacted chain corresponds to a genomic fragment and can be succinctly represented as a sequence of characters, often referred to as a unitig or a contig depending on the assembler used. Unitigs are used in subsequent assembly steps

to form longer sequences, and can significantly reduces the size of the de Bruijn graph [7] for later assembly steps.

Chain compaction is a significant performance bottleneck in the assembly process [43, 65]. Georganas *et al.* [63] reported that unitig generation, for which graph compaction is a significant component, accounted for $58\%$ and $78\%$ of the total time for assembling the human and wheat genomes respectively using the sequential Meraculous assembler. Their work then focused on distributed memory parallelization of the construction and chain compaction of de Bruijn graphs with significant improvements, such that scaffolding has become the primary contributor of running time.

As chain compaction establishes the traversal order of chain vertices, it is related to the classic list ranking problem: given a set of nodes that form a linked list and pointers to their neighbors, compute the distance of each node from the end of the list. However, chain compaction requires simultaneous identification and compaction of all chains in the graph, while differentiating cycles. The presence of a cycle will cause a naïve execution of the list ranking algorithm to enter an infinite loop. Additional complexities arise for bi-directed de Bruijn graphs, requiring special handling of bi-directed edges to avoid circular traversal and inadvertent backtracking [66, 62, 61]. Nevertheless, assemblers often directly leverage or conceptually adopt list ranking.

Sequentially, list ranking can be trivially computed, whereas parallel list ranking has been a research topic in its own right. The first parallel list ranking algorithm was proposed for the PRAM model by Wyllie [67], using $N$ processors and $\log N$ iterations for a list of size $N$, with total work of $N \log N$. Wyllie's algorithm relies on the strategy of *pointer jumping*, where the pointer of each list node is updated to its neighbor's pointer in each iteration. Subsequent shared memory algorithms were made work-optimal ($O(N)$ work) using strategies including *independent set removal* and *sparse ruling set*. In independent set removal, a random or deterministic subset of list nodes is selected and list ranked, then the remaining list nodes are reinserted and the ranking updated. With sparse ruling set,

multiple invocations of sequential list ranking are initiated from randomly selected "ruler" nodes, and the sub-lists are merged when ranking encounters other ruler nodes.

These strategies were evaluated and summarized by Reid-Miller and Blelloch [68]. Subsequent shared memory algorithms reduced the total work to the optimal $O(N)$ using additional strategies including independent set removal and sparse ruling set [69, 70, 68]. Algorithms designed for distributed memory parallel list ranking follow the same basic strategies [71, 72]. Dehne and Song [71] used randomized selection to produce independent sets, while Sibeyn *et al.* [72] optimized a parallel list ranking implementation, incorporating all three strategies, for the Intel Paragon system.

Parallel assemblers and tools employ primarily pointer jumping and sparse ruling set approaches for chain compaction. YAGA [61] and SWAP [62] explicitly adopt list ranking with pointer jumping. YAGA treats the bi-directed graph as directed, while SWAP tracks the directions of traversal during list ranking.

ABySS [8], HipMer [63, 9] implicitly use the sparse ruling set strategy by choosing random vertices as the the ruling set and "growing" each chain vertex by vertex. Zeng's algorithm [65] chooses end-of-chain vertices as the ruling set, while PASHA [43] used a combination of both end-of-chain and random vertex selection.

ParBiConstruct [66] and BCALM 2 [59] support only the graph construction and chain compaction steps. ParBiConstruct splits the vertices to transform the bi-directed graph into a directed graph, then explicitly uses list ranking with pointer jumping, similar to YAGA and SWAP, in distributed memory environments. BCALM 2 uses lexicographically minimal substrings within $k$-mers to partition the $k$-mer set. Its chain compaction process therefore bears resemblance to the sparse ruling set approach. Except for YAGA and BALM 2, these parallel assemblers require frequent fine grained communication.

While most of these assemblers support bi-directed de Bruijn graphs, where the double stranded nature of DNA is explicitly modeled, they require special handling, such as edge splitting [66, 62], to avoid circular traversal and inadvertent backtracking when processing

bi-directed edges and vertices.

*Circular Sequence Identification*

Cycles arise naturally in genomic data, and are handled differently by each chain compaction and assembly tool. Georganas [63] and Zeng [65] traverse chains in parallel, but extend each chain one vertex at a time. Both employ binary flags to indicate whether vertices were previously visited thus avoiding circular traversals. However, this approach cannot distinguish nor separate cycle vertex from chain vertices where multiple processors may concurrently traverse a chain.

Kundeti [66] relies on strictly decreasing number of merged tuples, referred to as *join count*, as tuples less than $2^t$ away from a chain terminal do not need to be merged further after iteration $t$. The algorithm terminates when the join count remains the same for two consecutive iterations.

BCALM2 [59] explicit excludes cycles from its algorithmic discussion for the rationale that circular sequences cannot be represented in its output.

### 1.1.3 Error detection and de Bruijn Graph Simplification

Graph simplification is useful for identifying and removing erroneous $k$-mers and resulting erroneous local structures in the graph. A simple and ubiquitously implemented approach is to filter the input $k$-mers by the number of their occurrence. This relies on relatively high NGS sequencing coverage and low sequencing error rate, and is more difficult to apply to sequencing output of non-uniformly sequenced genomes and in the case of metagenomics, rare species. This approach is use by all surveyed software and algorithms.

In de Bruijn graphs, erroneous structure can also be identified by inspecting local graph structures and associated metadata, including $k$-mer frequencies. These include bubbles, dead-ends, and spurious links. Their identification, removal, and correction can be useful for *de novo* genome assembly, and is implemented in Velvet, ABySS, and IDBA.

SAGE [73], and HINGE [74] also explicitly identify and correct dead-ends and bubbles but on string graphs and overlap-layout-consensus graphs. Most de Bruijn graph based assemblers therefore implicitly rely on the gap-closing step to avoid traversing the erroneous structures.

## 1.2 Research Objective

A survey of existing tools and algorithms for bioinformatic sequence analysis, specifically $k$-mer indices and de Bruijn graph implementations leads to several important observations.

First, the diversity of available algorithms and implementations affirms the importance of $k$-mer indexing and counting as a foundational capabilities for bioinformatics applications and analysis tasks. The multitude of genome assemblers similarly indicates the centrality of *de novo* genome assembly for bioinformatics sequence analysis.

Second, there is a distinct lack of reusable API and library implementation for integration into applications. Most of the surveyed algorithms and tools have implementations that are focused on standalone invocation via a command-line interface, or integrated into a specific application pipeline. SeqAn [75] represents an exception in this arena.

Third, efficient distributed memory parallel algorithms and implementations for $k$-mer counting and indexing are practically non-existent, while distributed memory parallel de Bruijn graph data structures and operations are closely embedded in each assembler, such that efficient reuse in other applications is difficult.

Finally, the genome assembler and unitig generation implementations reviewed shared similar processing stages including construction, error removal, chain compaction, unitig generation, and scaffolding.

The combination of the second and third observations implies that it would be inefficient and potentially difficult to reuse existing tools, including $k$-mer counter and indices, especially in a distributed memory application where tightly coupled communication and computation are necessary.

The observed common pipeline tasks for assembly suggests that abstracting and modularizing a standard set of tasks and associated primitive operations, such as graph query, traversal, and filtering, is desirable and feasible. They serve as parallel building blocks for current and future applications. Interestingly, little research effort has been previously devoted to abstracting, modularizing and standardizing such primitive operations and tasks.

Common to both $k$-mer indexing tools and de Bruijn graph implementations are therefore the scarcity of reusable building blocks and even more so those with distributed memory system support. To address the big data challenge for biological sequence analysis, it is increasingly necessary to leverage highly scalable distributed memory systems such as clusters and supercomputers with expanded memory capability and computational resources. High performance parallel $k$-mer indices and de Bruijn graph data structures and associated graph operations can significantly improve the performance and scalability of sequence data analysis, genome assembly and provide a common platform on which to develop solutions to bioinformatics problems using highly optimized and mature $k$-mer index and de Bruijn graph operation implementations.

The research presented embodies three high level aims:

1. Create the first reusable, flexible, and extensible distributed memory parallel libraries, with simple API and optimized implementations, for $k$-mer indexing and de Bruijn graphs for bioinformatics applications.

2. Develop efficient data structure and algorithms specifically for distributed memory parallel $k$-mer counting and indexing

3. Develop efficient data structure for distributed memory parallel de Bruijn graphs, and efficient algorithms for graph construction, compaction, traversal, and error removal.

Our contributions from this research includes efficient distributed memory algorithms for $k$-mer counting and indexing as well as de Bruijn graph construction, compaction, and error removal. Their implementations are presented in two parallel libraries, Kmerind and

Bruno, that are generic, flexible, and reusable. Our evaluations on distributed memory clusters as well as on large memory multi-core systems demonstrate their performance and scalability against existing state-of-the-art $k$-mer counters and de Bruijn graph compaction tool and assembler. In addition, we developed architecture-aware optimizations for hash functions and hash tables that significantly improves the performance of operations in the Kmerind library. Finally, we note that while our target domain is bioinformatics, we approached algorithm design and implementation with the aim for broader applicabilities in computer science and other application domains.

The dissertation is organized as follows. In Chapter 2 we present Kmerind, a distributed memory $k$-mer indexing and counting library. We describe its API design philosophy, as well as the data structure, distributed memory algorithms, and implementation. We then evaluate its performance in distributed and shared memory settings, and compare it to existing, state-of-the-art counting tools.

In Chapter 3 we explore performance optimizations of $k$-mer counting under Kmerind, including vectorization of hash functions, cache friendly and memory access efficient hash table algorithms, and MPI communication and computation overlap strategies,

In Chapter 4 we describe the Bruno library's design. We also present the graph construction algorithm and discuss in detail the chain compaction and unitig generation algorithms as well as cycle identification approaches. We conclude the chapter with a description of the 3 levels of graph operations supported by Bruno.

In Chapter 5 we present algorithms for detecting and removing errors based on occurrence frequency and local graph topology. We begin with a description of the type of errors and the approach to filter ($k$ + 1)-mers prior to construction. The algorithms for identifying bubbles and dead-ends are then presented, followed by a description of chain re-compaction to merge chains after graph modifications.

In Chapter 6 we present performance evaluation results for all de Bruijn graph operations in Bruno, namely graph construction, chain compaction, and graph cleaning. We

present the results of performance, scalability and parameter evaluations, and compare the performance of compaction and unitig generation to those of the state-of-the-art distributed memory assembler and unitig generation tool.

Finally, in Chapter 7 we briefly summarize our contributions, applicability of our algorithms and implementations beyond bioinformatics, and future areas of research and development.

# CHAPTER 2

# $K$-MER COUNTING AND INDEXING

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

In Kmerind, these objectives are directly reflected by our algorithms, APIs, and implementation. Shared-memory computers are inherently limited by computational resources including the number of processor cores and size of memory. We target distributed-memory environments to support scaling to very large data sets, utilizing very large amounts of memory, and recruiting substantial extra computational resources when performance is paramount. Kmerind classes and functions are written as C++ templates, thus enabling convenient creation of application-specific indices and easy customization of $k$-mer length, alphabet, and functions associated with index operations. All Kmerind indices are built upon the same set of basic operations, i.e. `insert`, `find`, `count`, and `erase`, with sequential semantics and parallel implementations. The target users, application developers, can readily extend the capability of our library by adding new algorithms or optimizing the implementations of existing components.

## 2.1 Distributed Memory Algorithms

Kmerind's algorithms are designed to be efficient in both computation and communication complexities. We leverage bulk-synchronous parallel communication primitives with explicit synchronization semantics to enforce coarse grain synchronization that avoids contention, reduces overhead, and encourages communication optimization by MPI. Specifically, Kmerind algorithms employ primitives defined in version 2 of the Message Passing Interface (MPI) standard [76]. Kmerind does not use multithreading as thread-safety incurs additional overheads for this highly data parallel task. To minimize costly file system access, Kmerind indices are memory resident for the duration of their use.

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

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

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

### 2.1.1 Distributed $k$-mer Parsing

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

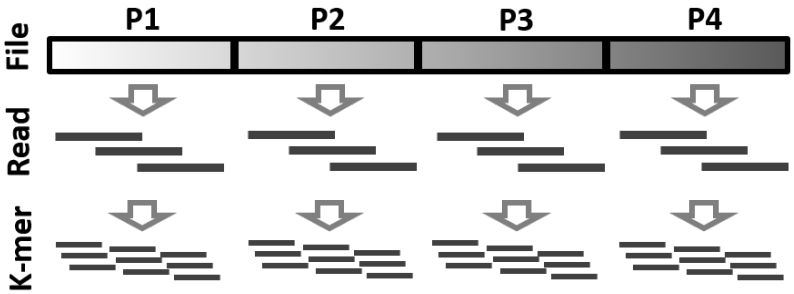



Figure 2.1: Parallel sequence file reading in distributed environment.

Parallel file reading and $k$-mer parsing proceeds in 3 steps, *file partitioning*, *sequence segmentation*, and *$k$-mer generation*, as shown in Figure 2.1 and Algorithm 2.1. Parallelization circumvents single machine limits. Each step maintains load balance across processors and linear time complexity.

During *file partitioning* (Algorithm 2.1 lines 2–3), $F$ is divided into approximately equal partitions of $|F|/p$ bytes and loaded in parallel into main memory as a character array $S$. The *sequence segmentation* step (Algorithm 2.1 lines 5, 6, 15) employs linear-time, format-specific logic to identify in $S$ individual sequences, from which $k$-mers, positions, and other data are extracted. For each segmented sequence, the *$k$-mer generation* step (Algorithm 2.1 lines 8–14) extracts $k$-mers via a sliding window and stores the results in an array to be used as input for the distributed index operations. When an unknown character is encountered, one of three strategies is used: discard the sequence; discard the $k$-mer; or replace the unknown with a known character such as "N".

**Algorithm 2.1** Distributed $k$-mer Parsing

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

*FASTQ Sequence Segmentation*

Kmerind currently supports the common FASTQ format where each sequence record has the form:

```
@{sequence identifier}

ATCG...

+{description}

quality score
```

A partition boundary may fall within a sequence record, in which case the record is assigned wholly to the first processor, while the second processor's partition is shifted to align with the first complete record. As the *quality score* line may begin with `@` or +, identifying the start of a record requires a $4$-line context. During `InitSequenceSegmentation` (Algorithm 2.1 line 5), each processor records the positions and first characters of the first $4$ complete lines in $S$ and calculates the starting position via Table 2.1. Partial record before

the start of a partition is sent to the previous processor via `MPI_Send`. We note that it is unlikely for a record to span more than two processors, as FASTQ records are significantly shorter than partition size.

Table 2.1: Lookup table for determining the position of the first complete record in a file partition.

| First Character | | | | |
|---|---|---|---|---|
| Line 1 | 2 | 3 | 4 | Record Starts on |
| @ | | + | | Line 1 |
| | @ | | + | Line 2 |
| + | | @ | | Line 3 |
| | + | | @ | Line 4 |

Once the partitions are aligned to start on the first full sequence record on each processor, sequences can be trivially segmented iteratively during `GetNextSequence` (Algorithm 2.1 lines 6,15) by reading the next $4$ complete lines from $S$. User supplied logic may simultaneously compute other data including position and $k$-mer quality score.

*FASTA Sequence Segmentation*

Kmerind currently supports multi-FASTA format where each sequence record has the form:

```
>{sequence identifier}
ATCG...
GCTA...
```

As FASTA sequences in a file can vary greatly in number and length, FASTA file parsing in Kmerind maintains exact block partitioning of size $|F|/p$ with ($k$-1)-byte overlap to promote load balance. During `InitSequenceSegmentation` (Algorithm 2.1 line 5), each partition is scanned linearly and the starting and ending positions of each sequence record are stored in a local array for application usage, such as generating $k$-mer positions. Global sequence identifiers are computed via parallel prefix sum.

A FASTA sequence, like FASTQ sequence, may be split between file partitions. However, since a FASTA sequence may contain a complete genome, it is more likely to span multiple partitions. A partial sequence containing the header marker ">" is referred to as *leading*. The global identifiers and starting positions of the *leading* partial sequences are propagated to the remaining partial sequences by initializing the identifiers and starting positions of non-leading partial sequences to $0$, followed by updating using exclusive parallel prefix scans with the $max(\cdot)$ operator. The ending positions are similarly propagated via reverse parallel prefix scan with the $min(\cdot)$ operator. Complete sequences require updates only to their global identifiers. We note that each partial sequence remains on the original processor.

Sequences are then trivially segmented by iterating over the previously computed array of sequence identifiers and starting and ending positions, and extracting the corresponding characters from $S$, during `GetNextSequence` (Algorithm 2.1 lines 6,15).

*Complexity Analysis*

We denote the average sequence record length, including header and other data, as $L$. We assume that that the time to process FASTA partition overlaps is negligible as $k \ll |F|/p$.

The *file partitioning* step (Algorithm 2.1 lines 2–3) requires $O(|F|/p)$ space and time for reading the file partition for both FASTA and FASTQ files. The *$k$-mer parsing* step (Algorithm 2.1 lines 8–14) requires linear time and space, $O(|F|/p)$, for parsing and storing $k$-mers. No inter-processor communication is required.

For FASTQ files, the *sequence segmentation* step requires $O(L)$ time for scanning for the offset of the first complete sequence. Once found, moving the partial sequence to the previous processor requires at most $O(\tau + \mu L)$ communication time and $O(L)$ additional space. Since $L \ll |F|/p$, this step has a negligible contribution to the overall time. Parsing each sequence requires $O(L)$ time. Parsing all sequences thus results in a complete linear scan of the file partition in $O(|N|/p)$ time, constant space, and no communication.

For FASTA files, the *sequence segmentation* step initialization requires $O(|N|/p)$ time to parse all sequences and $O(|N|/pL)$ space for storing the resultant objects. Updating the partial sequences with global information involves 3 prefix scans over the first and last sequence objects from each processor, thus requiring $O(\tau \log(p) + \mu \log(p))$ communication time, which are negligible as $\log(p) \ll |N|/p$. Iterating over all sequence objects during $k$-mer parsing incurs constant time and space overheads and no communication per iteration.

For both FASTA and FASTQ file parsing, the overall time and space complexity is $O(|N|/p)$, and there is a negligible communication cost in both cases. We note that file system performance can significantly affect the *file partitioning* step. Local file system performance can benefit from the use of solid state drives (SSDs), while parallel file system performance depends significantly on network bandwidth and file system configurations. System level file caching also can have a strong impact on file partitioning performance.

### 2.1.2 Distributed Indices: Overview

Kmerind's distributed $k$-mer indices are modeled as either hash maps or sorted arrays of $k$-mers and associated data. In both cases the $k$-mer space is partitioned amongst the processors so that an input $k$-mer for an index operation is deterministically assigned to and processed by one and only one processor. We also considered look-up tables and suffix arrays. While a distributed look-up table can enumerate the entire $k$-mer space as a $4^k$ array for small $k$, non-uniformity in $k$-mer distribution of the genome or read file can translate to computational load imbalance. Suffix arrays and trees, while flexible and memory efficient, are not well suited for distributed-memory queries as stated in Section 1.1.

Kmerind's distributed **hashed** index is designed as a two-level hash map. The upper level hash function maps $k$-mers to processor ranks uniquely and deterministically, while the lower level consists of a local hash map for storing $k$-mers and associated data. Hash function for each level is user definable and should be chosen to produce (1) uniformly dis-

tributed hash values to ensure load balance across processors and minimize hash collisions within local hash maps, and (2) uncorrelated upper and lower level hash values to avoid clumping, where $k$-mers mapping to the same processors are assigned to the same map buckets.

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

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

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

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

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

than $|N_i|$, $|M_i|$, and $|R_i|$.

### 2.1.3 Distributed Hashed Index

For each operation in a distributed hashed index, the input data is first assigned (`map_to_processor`) then communicated to the target processor (`distribute`). The communicated data is then processed locally on each processor, with results optionally communicated back to the source processor. Figure 2.2 illustrates this process for insertion.

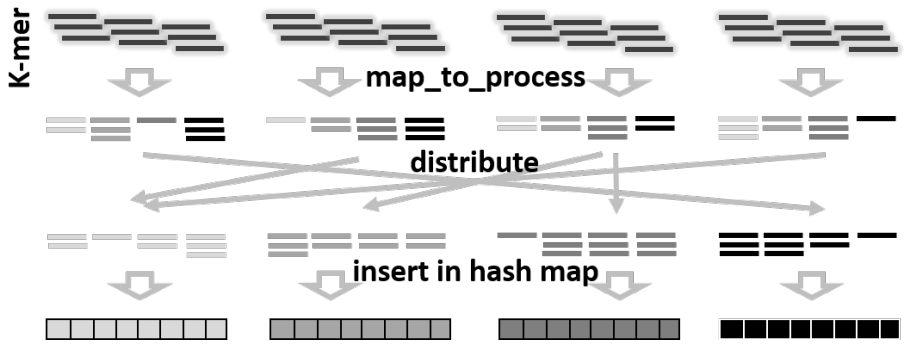



Figure 2.2: Inserting $k$-mers into the Kmerind's distributed index.

*Data Movement*

Input data movement for an operation of a distributed hashed index begins with the `map_to_processor` operation outlined in Algorithm 2.2, where, `mapper` is the upper level hash function. The $k$-mers are first assigned to target processors using the upper level hash function (Algorithm 2.2 lines 4–7). Both the assignments and the number of $k$-mers in each bucket are saved simultaneously. The $k$-mers in the input array are then stably permuted (Algorithm 2.2 lines 14–17) so that $k$-mers for the same processor occupy contiguous memory as required by the communication primitives. The bucket assignment array is converted into a *source-to-target* position mapping array (Algorithm 2.2 lines 8–13). The position mapping array is used during permutation, and for inverse permutation if input data in the original order is needed.

**Algorithm 2.2** Mapping $k$-mers to processors

```
1: function MAP_TO_PROCESSOR(M_i, mapper, p)
2:     map ← array of size |M_i|
3:     counts ← array of size p
4:     for i ← 0, (|M_i| − 1) do
5:         map[i] ← mapper(M_i[i]) mod p
6:         counts[map[i]] ← counts[map[i]] + 1
7:     end for
8:     offsets ← exclusive_prefix_sum(counts)
9:     for i ← 0, (|map| − 1) do
10:        rank ← map[i]
11:        map[i] ← offsets[rank]
12:        offsets[rank] ← offsets[rank] +1
13:    end for
14:    M'_i ← array of size |M_i|
15:    for i ← 0, (|M_i| − 1) do
16:        M'_i[map[i]] ← M_i[i]
17:    end for
18:    return ⟨M'_i, counts, map⟩
19: end function
```

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

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

The complexity of the `distribute` function depends directly on the space and time complexities of MPI operations, $O(\tau \log(p) + \mu |M_i| \log(p))$ time and $O(|M_i| + |_r M_i|)$

space, linear in the size of the input and output.

*Distributed Hashed Uni-Index*

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

**Algorithm 2.3** Distributed Hashed Index `insert`

```
1: procedure INSERT(M_i, C, mapper, p)
2:     ⟨M'_i, counts⟩ ← map_to_processor(M_i, mapper, p)
3:     _rM_i ← distribute(M'_i, counts)
4:     for x ∈ _rM_i do
5:         v ← C.insert(x)
6:     end for
7: end procedure
```

The algorithms for `insert` and `count` index operations are shown in Algorithm 2.3 and Algorithm 2.4, respectively, where $M_i$ is an array of query $k$-mers on processor $i$, C is the local container, and $R_i$ contains the query results for processor $i$. In each algorithm, the input is assigned and distributed to the target processors using `map_to_processor` and `distribute` operations as described in Section 2.1.3. The target processors then process the received $k$-mers and data locally (Algorithm 2.3 line 5, Algorithm 2.4 line 5). The `count` algorithm returns the count results to the source processors via collective communication using the `inverse_distribute` operation.

The algorithms for `erase` and `find` are essentially identical, except the local hash map operations at Algorithm 2.3 line 5 and Algorithm 2.4 line 5 are replaced with C.`erase` and C.`find`, respectively. Duplicates in input may be removed before invoking the hashed index operations to reduce subsequent communication volume and computational load.

**COMPLEXITY ANALYSIS**: We assume appropriate hash functions were chosen so that data is distributed as uniformly as possible, $|M_i| \approx |_rM_i|$, and that the local hash map has

**Algorithm 2.4** Distributed Hashed Index `count`

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

expected $O(1)$ access time per $k$-mer.

*Update* operations `insert` and `erase` requires $O(|M_i|)$ space and time complexity for `map_to_processor` (Algorithm 2.3 line 2, Algorithm 2.4 line 2), $O(|M_i| + |{}_rM_i|)$ space and $O(\tau \log(p) + \mu |M_i| \log(p))$ time for `distribute` (Algorithm 2.3 line 3, Algorithm 2.4 line 3), and $O(|{}_rM_i|)$ time and space for local hash map insertion and erasure (Algorithm 2.3 line 5, Algorithm 2.4 line 5). *Update* operations have overall complexity of $O(|M_i| + |{}_rM_i|)$ in space, and $O(\tau \log(p) + \mu |M_i| \log(p))$ in communication time and $O(|M_i| + |{}_rM_i|)$ in computation time.

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

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

*Distributed Hashed Multi-Index*

Kmerind's hashed multi-index uses a local hashed multi-map to associate multiple values to each $k$-mer. The local hashed multi-map implementation can affect the per-element access time complexities, however. Kmerind's choice of local hash multi-map is described in Section 2.2.2. Here we assume that time complexity is expectedly $O(1)$ for each local multi-map access.

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

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

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

The second order dependence of $|_rR_i|$ on $r_i$ implies that non-uniformity in frequency distribution can quickly cause load imbalance in computation, memory usage, and communication for the uni-index *query* algorithm Algorithm 2.4.

Instead, Kmerind's position index's *find* operation uses Algorithm 2.5 that amortizes the space and time requirements over $p$ iterations. During each iteration, the query $k$-mers from one source processor are processed. The query $k$-mers are first assigned and distributed

**Algorithm 2.5** Distributed Hashed Multi-Index `find`

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

(Algorithm 2.5 lines 2–3). The total number of result tuples are counted and returned to the source processor (Algorithm 2.5 lines 4–11) so that the result array $R_i$ can be allocated (Algorithm 2.5 lines 12–13). We then iterate over each query $k$-mer subsets $M_{j \to i}$ by processor rank $j$ (Algorithm 2.5 lines 14–23), finding all results for a subset (Algorithm 2.5 line 19) and sending the subset result immediately (Algorithm 2.5 line 21) before processing the next subset. Non-blocking point-to-point communication (`MPI_Isend` and `MPI_Irecv`) is used, which allows communication to overlap query processing computations.

**COMPLEXITY ANALYSIS**: Distributed hashed multi-indices have identical complexities for the `insert`, `erase`, and `count` operations as those for the uni-indices (Section 2.1.3).

The `find` algorithm for the multi-index aims to minimize the intermediate memory

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

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

### 2.1.4 Distributed Sorted Index

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

#### *Data Movement*

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

We adopt hashed index's `map_to_processor` algorithm (Algorithm 2.2), with the exception that line 5 is replaced with a binary search for the query $k$-mer's insertion position in the *splitter* array. The insertion position corresponds to the processor rank to which the $k$-mer should be sent.

Subsequent to mapping, the query $k$-mers can be sent to the target processors via the `distribute` and `inverse_distribute` operations from Section 2.1.3.

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

*Distributed Sorted Uni- and Multi-Indices*

Data movement in Section 2.1.4 depends on the presence of the *splitter* array, which is constructed during or after parallel sorting. The `insert` operation for distributed sorted uni-index and multi-index employs parallel sample sort [77] with regular sampling to sort the input $k$-mer array and produce the *splitter* array concurrently.

The `erase`, `count` and `find` operations for both the sorted uni- and multi-indices employ the same algorithms as those for the hashed indices: Algorithms 2.3, 2.4, and 2.5. The hashing `map_to_processor` operations (Algorithm 2.3 line 2, Algorithm 2.4 line 2, Algorithm 2.5 line 2) are replaced with the binary search version described in Section 2.1.4. The local hash map `erase` (Algorithm 2.3 lines 4–6), `count` (Algorithm 2.4 lines 4–7), and `find` (Algorithm 2.5 lines 18–20) operations are likewise replaced with their sorted array counterparts.

During the local `erase` operation (Algorithm 2.6), the query $k$-mers ${}_rM_i$ are first sorted so that binary search ranges in the indexed array can be reduced successively. For each query $k$-mer, the matching range in the sorted array is identified (Algorithm 2.6

**Algorithm 2.6** Local `erase` for Sorted Index

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

lines 7,12) then overwritten in the next iteration with the array elements *between* successive matching ranges (Algorithm 2.6 lines 8–11, Algorithm 2.6 lines 14–17). After all query $k$-mers are processed, the remaining non-matching array elements are moved forward. The sorted array size is thus reduced.

**Algorithm 2.7** Local `count` for Sorted Multi-Index

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

The local `count` algorithm (Algorithm 2.7) similarly sorts the query $k$-mers first. For each $k$-mer in the query set, the matching range is computed via 2 binary searches (Al-

gorithm 2.7 lines 7–8), then the count is computed from the range (Algorithm 2.7 line 9). The local `find` algorithm differs only in that elements in the round range are copied to $R_i$ instead of computing the count (Algorithm 2.7 line 9).

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

The `insert` operation for Kmerind's distributed sorted indices has time complexity equal to that of parallel sample sort [77]. The computation time is dominated by local sorting $O(|M_i| \log(|M_i|))$ assuming total sample size is negligible compared to input data size, $p^2 \ll |M_i|$, while communication complexity is $O(\tau \log(p) + \mu(p + |M_i|) \log(p))$. Space required is primarily for communication buffers thus linear in input size, $O(|M_i|)$.

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

For the `erase` operation, the computation complexity is $O(|M_i| \log(p) + |_r M_i| \log(|N_i|) + |N_i|)$. The first term is for assigning query $k$-mer to processors, the second for searching for matching $k$-mers, while the third is for moving non-matching $k$-mers. The `count` operation has similar computation complexity, with the last term being $|_r M_i|$ for computing the size of the matching range for each query $k$-mer. For for both uni- and multi-indices, the last term for the `find` operation is $r_i |_r M_i|$, due to copying of all elements in the matching range.

## 2.2 Implementation

We designed and implemented Kmerind as a distributed memory parallel library based on the objectives listed in Section 1. Kmerind is a header only C++ library with a tiered

architecture (Figure 2.3). It leverages C++ 11 language features, Standard Template Library (STL) containers and algorithms, and MPI and the *mxx* [78] MPI wrapper library. Each tier defines templated functions and class interfaces as well as default implementations to allow functionality by composition and extension by specialization and inheritance, thus providing Kmerind's flexibility and extensibility.

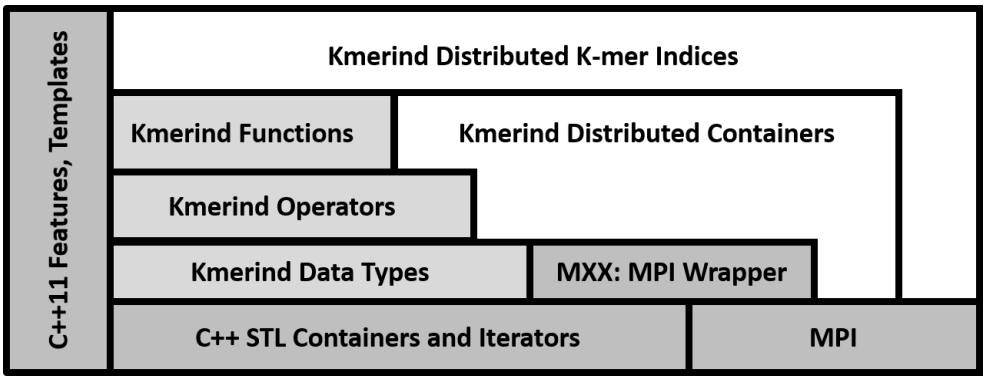



Figure 2.3: Kmerind Library's tiered architecture.

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

The *Distributed $k$-mer Indices* layer contains $k$-mer indices, which are implemented as light-weight wrappers for *Distributed Containers*. The distributed containers implement algorithms described in Sections 2.1.3 and 2.1.4, and use local hash maps or sorted arrays for local storage. Additionally, Kmerind provides maps that perform reduction on insertion, an example being a counting map.

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

### 2.2.1 K-mer Representation

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

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

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

Rather than explicitly model double stranded $k$-mers, Kmerind accounts for the double stranded nature in the indexing operations. Kmerind indices can manage and query each $k$-mer as-is (single strand mode), convert each $k$-mer to canonical (canonical mode), or store $k$-mer as is but accept $x$ and $\bar{x}$ as equivalent for queries (bimolecule mode). In bimolecule mode, an index's hash and comparison functions compute $\tilde{x}$ on demand.

#### *SIMD Friendly K-mer Representation*

The performance of $k$-mer reverse complement operation `revcomp` is critical and has been vectorized using Single Instruction Multiple Data (SIMD) hardware instructions and SIMD Within A Register (SWAR) [79] patterns where only x86 instructions are used.

The `revcomp` operation proceeds in two conceptual phases: character order reversal and character complement. To reverse the order of characters, each word in a $k$-mer is

byte-reversed, and each byte is then character-reversed. Machine words in a $k$-mer are processed in linear order.

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

To accelerate character complement, the bit encoding of characters as defined by $\Sigma$ are chosen so that the complement of a character can be computed via simple vectorizable functions. For the DNA5 and IUPAC DNA, the complement function is *bit reversal*, while for the DNA alphabet, *bitwise negation* is used. Examples are given in Table 2.2.

Table 2.2: SIMD-friendly bit encoding for DNA, DNA5, and IUPAC DNA characters and corresponding character complement method. For IUPAC DNA alphabet, not all characters are shown.

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

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

### 2.2.2 Local Hash Table

Kmerind's distributed hash map implementation allows different local hash map implementations to be used. Kmerind incorporates Google SparseHash's Dense Hash Map (referred to as *DHM*) [80] as the default local hash table due to its performance. *DHM* uses open addressing with quadratic reprobing, thus requiring 2 dedicated keys to identify *empty* and *deleted* hash table slots. The choice of these keys depends on $k$-mer parameters and the strand mode of the index as defined in Section 2.2.1. Table 2.3 summarizes the decision tree for selecting the strategy to generate the *empty* key for *DHM*. *Deleted* key selection is similar.

Table 2.3: Strategies for choosing $k$-mers as the *empty* key for *DHM*. Conditions listed are checked successively row by row. If a condition is met, the strategy listed on that row is used. Examples shown are 3-mers in ASCII or binary encoding.

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

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

We extended *DHM* into a multimap in order to support distributed multi-maps. *Dense*

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

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

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

### 2.2.3 $K$-mer Count and Position Indices

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

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

## 2.3 Experimental Results

Table 2.4: Experimental data sets used for all evaluations. Where applicable, accession numbers for NCBI are provided.

| Id | Organism | Genome Size (Mbases) | File Format | File Count | File Size (Gbytes) | Sequence Count | Average Read Length | Source | Accession |
|---|---|---|---|---|---|---|---|---|---|
| R1 | H. sapiens | 2,991 | FASTQ | 1 | 6.3 | 23,861,612 | 101 | 1000 Genome HG00096, NCBI | SRR077487 forward only |
| R2 | F. vesca [81] | 214 | FASTQ | 11 | 14.1 | 12,803,137 | 352 | NCBI | SRA020125 |
| R3 | G. gallus | 1,230 | FASTQ | 12 | 115.9 | 347,395,606 | 100 | NCBI | SRA030220 |
| R4 | H. sapiens | 2,991 | FASTQ | 48 | 424.5 | 1,339,740,542 | 101 | NCBI | ERA015743 |
| G1 | H. sapiens | 2,991 | FASTA | 1 | 2.9 | 84 | – | 1000 Genome reference GRCh37 | – |
| G2 | P. abies [82] | 20,000 | FASTA | 1 | 12.4 | 10,253,694 | – | Congenie.org | – |
| M1 | metagenome | – | FASTQ | 1 | 7.6 | 33,195,888 | 101 | IOWA Continuous Corn Soil ( Project 402461 ), Joint Genome Institute | – |
| M2 | | | | 1 | 15.2 | 66,391,776 | | | |
| M3 | | | | 1 | 30.4 | 132,783,552 | | | |
| M4 | | | | 1 | 60.8 | 265,567,104 | | | |
| M5 | | | | 1 | 121.6 | 531,134,208 | | | |

We examined the performance of Kmerind and compare it to those of a select subset of existing $k$-mer indexing tools. Data sets used for the experiments are summarized in Table 2.4, and referenced by "Ids" in subsequent discussions.

Sequential and multi-threaded tests were conducted on the CompBio system at Georgia Institute of Technology. CompBio contains four $2.1$GHz Intel Xeon E7-8870v3 processors with $45$MB L3 cache, $1$TB of DDR4 RAM, and RAID 1 file systems with rotating disks. All tested software were compiled with GCC v5.3 and OpenMPI v1.10.2 if required.

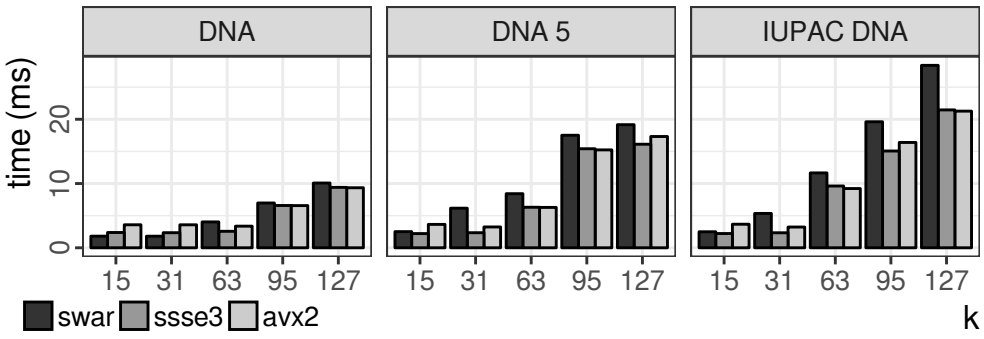
Distributed-memory experiments were conducted on Iowa State University's CyEnce cluster. Each node contains two $2.0$GHz $8$-core Intel Xeon E5-2650 CPUs and $128$GB of RAM. The cluster has quad data rate (QDR) Infiniband interconnect and is supported by a $288$TB Lustre file system with $1$ MDS and $8$ OSTs. All data files are stored on Lustre with 1MB block size and stripe count of $8$. All test binaries were compiled with GCC v$4.9.3$ and MVAPICH v$2.1.7$.

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

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

### 2.3.1 K-mer Operations

We benchmarked the SIMD accelerated $k$-mer reverse complement operation using the CompBio system. Figure 2.4a summarizes the times to compute reverse complement of one million DNA, DNA5, and IUPAC DNA $k$-mers for varying $k$ using the x86 SWAR,




(a)

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

(b)

Figure 2.4: Time in milliseconds to reverse-complement one million $k$-mers of varied alphabets using SWAR, SSSE 3 or AVX 2 instructions for different $k$ (a) and 31-mers (b).

SSSE 3, and AVX 2 `revcomp` implementations. Table 2.4b summarizes the times for 31-mers.

Overall the SIMD based `revcomp` implementation has a throughput of approximately two microseconds per 31-mer, and scales linearly with the number of machine words in the $k$-mer data structure. The computation time increases in fixed steps with word count instead of with $k$ directly. SWAR and SSSE 3 implementations were approximately $16\times$ and $21\times$ faster than sequential (SEQ in Figure 2.4b). AVX 2 performed comparably to the SSSE 3 for $k$ up to $256$ (data not shown) due to the additional overhead incurred when moving bits between 128-bit data lanes.

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

### 2.3.2 Distributed $k$-mer Parsing

Distributed file reading and $k$-mer parsing benchmarks were performed on the CyEnce cluster. Three different file access mechanisms were evaluated: MPI-IO, memory mapping (MMAP), and POSIX file access functions. Copies of the same files were used to isolate the effects of file system caching. Parallel $k$-mer parsing of the **R1** data set scaled nearly linearly for up to $p = 64$, beyond which the network was likely saturated (Figure 2.5). MPI-IO and MMAP mechanisms performed similarly given CyEnce's configuration. For 32 and 64 cores, the POSIX mechanism showed an approximately 40% advantage.

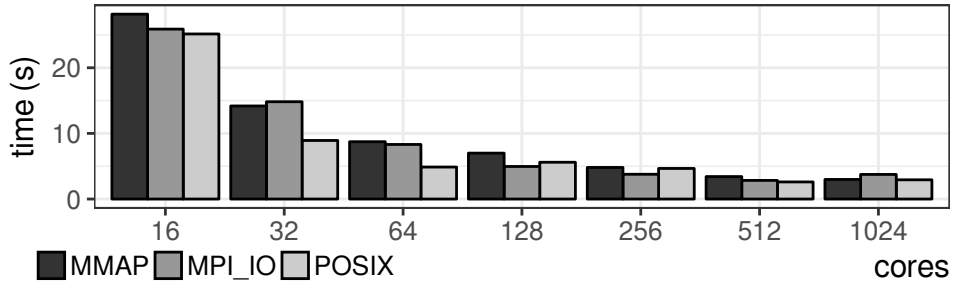



Figure 2.5: Time in seconds to read and parse the **R1** data set from disk into memory via MPI-IO, POSIX, and memory mapping, using varying number of cores. The $x$-axis is in logarithmic scale.

Table 2.5: Time in seconds to read a file from the Lustre file system using 128 cores, with and without operating system file caching.

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

The time to read and parse the **M4** data set using 128 cores was dramatically improved when the file was previously cached (Table 2.5). Different I/O mechanisms benefited from caching differently, with POSIX receiving a $22.8\times$ speed up. The long uncached file reading time and the short index construction time (Sections 2.3.4 and 2.3.5) suggest that rebuilding a $k$-mer index from cached sequence data is likely preferable to loading a previously built index.

In subsequent tests, we used POSIX and pre-populated file cache with a "warm up" iteration. File reading times were excluded from the index construction and query times for scalability experiments in Section 2.3.4, and included for comparisons to existing tools in Section 2.3.5.

### 2.3.3 Effects of Index Parameters

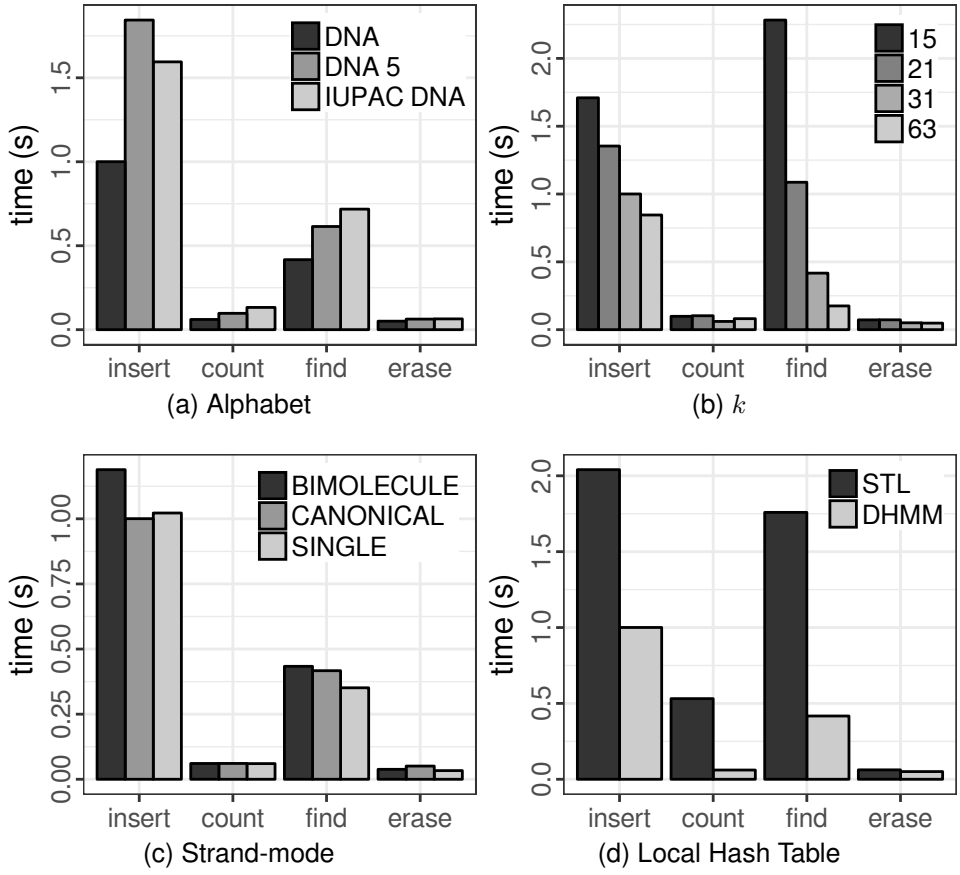



Figure 2.6: Parameters affecting the performance of Kmerind's indices. Experiments were conducted using 1024 cores on CyEnce and a Kmerind canonical hashing position index, configured with Google `farmhash` and DHMM, for DNA 31-mers in the **R1** data set.

Kmerind provides significant flexibility for users to configure the data structures and algorithms through parameters and compositions. In this section, we briefly examine some of the parameters and their impacts on performance. Figure 2.6 summarizes the index operation performance for $4$ common parameters. Alphabet, $k$, and *strand*-mode can be

considered as application-driven parameters, while the local map choice is performance-driven.

Among the 3 provided alphabets, DNA provides the best performance as it is compact and allows simple bitwise operations (Figure 2.6a) where DNA 5 requires a more complex character shifting algorithm during $k$-mer parsing from file. The bit length of encoded character is inversely related to the performance of the index, as is the value $k$. Each short read is parsed into $L - k + 1$ $k$-mers and large $k$ results in fewer $k$-mers, thus reducing running time (Figure 2.6b).

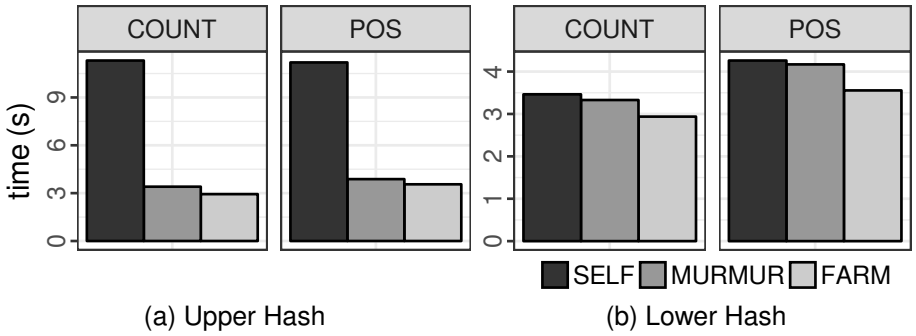



Figure 2.7: Hash function impact on performance of DNA 31-mer count and position indices. "SELF" indicates that the $k$-mer is used as hash value directly. "MURMUR" and "FARM" indicate MurmurHash3 and Google's farm hash, respectively. Data set **M3** was processed using 1024 cores.

Figure 2.6c shows the overhead for the *bimolecule* and *canonical* modes of operation. Bimolecule mode requires canonicalization for the input $k$-mers as well as the indexed $k$-mers, thus `revcomp` is applied at least twice for each $k$-mer during any operation. Canonical mode requires canonicalization once for each input $k$-mer, while *single-stranded* mode does not require any canonicalization, thus both perform better than the *bimolecule* mode.

For indices that use distributed hash map or multimap, different types of local hash map can be chosen. As shown in Figure 2.6d, DHMM consistently outperforms STL's *unordered_multimap* for all except for the `erase` operation. This is because unordered multimap requires the use of linked lists within each bucket.

The choice of hash functions for Kmerind's two-level distributed hash maps can have

a strong impact on the performance. We evaluated three hash functions, Google FarmHash (`farmhash`), MurmurHash3 (`murmur3`), and $k$-mer as hash value (`self`), on index insertion at the upper and lower hash levels. In each case, the hash function of one level is varied while the other level is set to use `farmhash`, The experiments were conducted using $1024$ cores and the **M3** data set in single-stranded mode for DNA $31$-mers.

Figure 2.7a shows that using `self` as upper hash function caused significant performance degradation when compared to the results from `farmhash` and `murmur3`. This degradation is attributable to severe load imbalance: while the standard deviations of the number of $k$-mers assigned to each core were $2,751$ and $2,806$ for `murmur3` and `farmhash` respectively, for `self` the standard deviation reached $6,662,250$. The standard deviations were $41,786$, $41,160$, and $7,710,595$ for `murmur3`, `farmhash`, and `self` based position indices, respectively. As points of reference, the average numbers of $k$-mers per core were $7,678,665$ for counting and $9,007,001$ for position indexing.

For the lower level hash function, hash collision and computational overhead are the primary concerns. Figure 2.7b shows that `self` and `murmur3` performed similarly for count and position indices, while `farmhash` outperformed both, likely due to more uniform hash value distribution than `self` and better computational efficiency than `murmur3`.

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

### 2.3.4 Scalability

In this section the scalability of each position and count index operation is examined (Figures 2.8, 2.9). In strong scaling experiments, the total input data set size $|M|$ is fixed while $p$ is increased to demonstrate an algorithm or software's capability of using additional re-

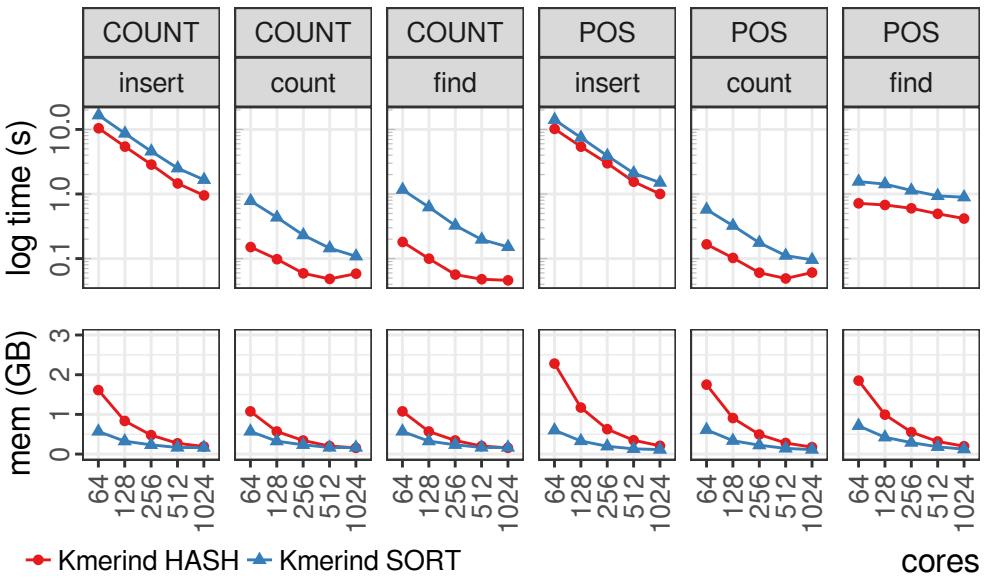



Figure 2.8: Strong scaling results for the `insert`, `count`, and `find` operations for the hashing and sorting variants of Kmerind's count and position indices, using canonical DNA 31-mers from data set **M1**.

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

Kmerind's hashed count and position indices ingested the **M1** data set in approximately $1$ second, and the **M5** data set in $12.9$ and $16.6$ seconds respectively using $1024$ cores. Retrieving the counts in the count index required $0.05$ and $0.23$ seconds for **M1** and **M5**. Retrieving the positions took $0.42$ and $28.0$ seconds for **M1** and **M5**, respectively.

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

**M5**, respectively.

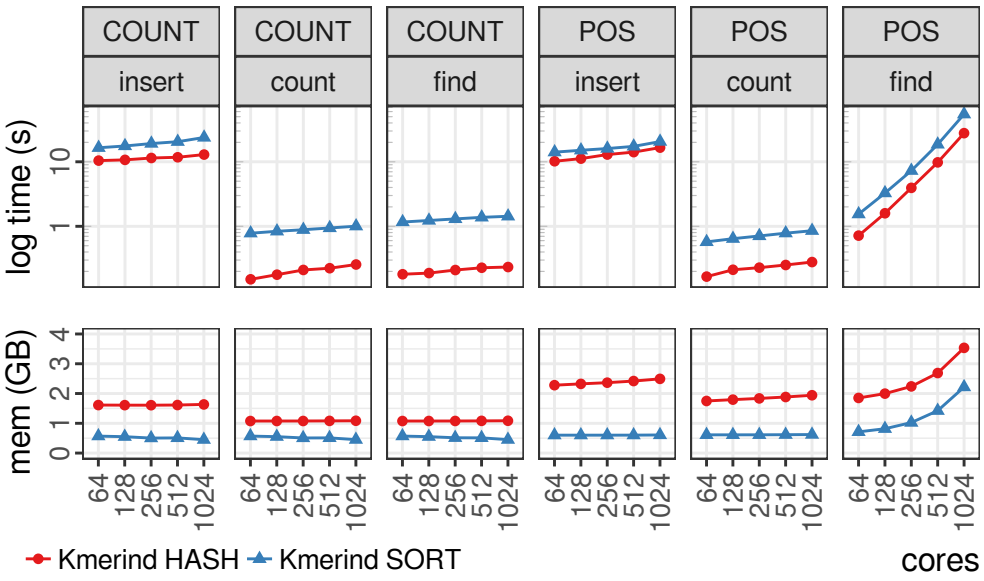



Figure 2.9: Weak scaling results for the `insert`, `count`, and `find` operations for the hashing and sorting variants of Kmerind's count and position indices, using canonical DNA 31-mers from data sets **M1**–**M5**.

Overall, the `insert`, `count`, and `erase` operations for both the hashed count and position indices showed similar scaling behavior. Similarly, sorted count and position indices exhibited same scaling trends for these 3 operations. The `find` operations for the corresponding count and position indices showed significantly different scaling behaviors as predicted in Sections 2.1.3 and 2.1.4. The `erase` operation times are not shown in Figures 2.8 and 2.9 as they closely mirror the scaling behavior of the `count` operation.

Figures 2.8 and 2.9 further illustrate the performance characteristics of Kmerind's sorted and hashed indices. Hashed indices consistently and significantly outperformed the sorted indices in time for the index operations, over $6\times$ faster for `find` operations on count indices. At the same time, sorted indices required significantly less memory during execution by nearly a factor of $4$ during count and position indices `insert`. This is particularly apparent in the weak scaling experiment results (Figure 2.9). The memory advantage of sorted indices decreased with core count for strong scaling as the overheads of local hash maps became less evident (Figure 2.8). Note that sorted indices are designed towards a balance

between lower memory requirement and acceptable performance and scaling, rather than minimal space requirement.

*Analysis of Scaling Behaviors*

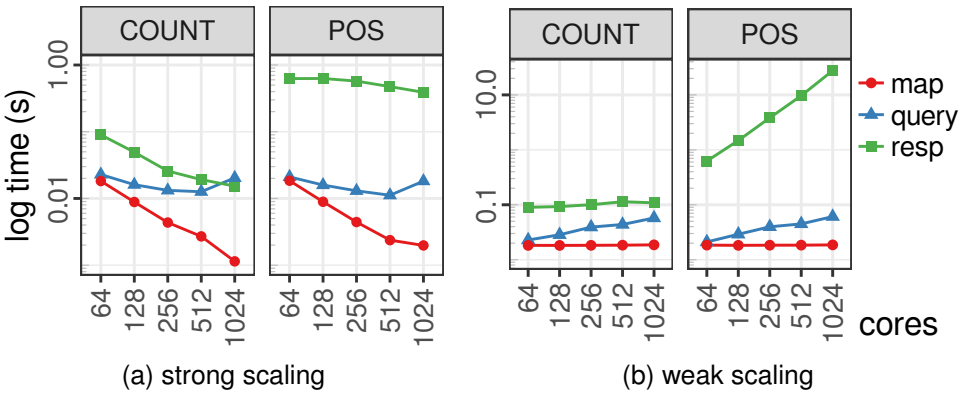



Figure 2.10: Strong and weak scaling of internal steps in the `find` operation for hashed count and position indices. The "map", and "query" steps correspond to the `map_to_processor` and `distribute` functions in Algorithms 2.3 and 2.5, while "resp" corresponds to all remaining algorithmic steps after the `distribute` step.

We examine the `find` operations of the hashed position and count indices in more detail to better understand the scaling behaviors that reflect the algorithmic and complexity differences described in Sections 2.1.3 and 2.1.3.

In strong scaling experiments, `find` for count index reached minima at 512 cores. Figure 2.10a shows that the presence of the minima is largely due to communication in the "query" and to a lesser degree to the "resp" steps with complexity $\tau \log(p) + \mu(|M|/p) \log(p)$. For strong scaling, as $p$ increases, the bandwidth term decreases at the rate of $\log(p)/p$, while the latency term increases at a rate of $\log(p)$. For large $p > \mu|M|/\tau$, latency dominates.

Scaling of the `find` operation for $k$-mer position index is dominated by the "resp" step with complexity $(r|M|/p) + \tau p + \mu(r|M|/p)$ (Section 2.1.3). In contrast to a count index, the "resp" step for the position index has significantly higher latency and computation complexities. In addition, the average $k$-mer frequency $r$ can increase the bandwidth term

contribution for $r > \log(p)$, and highly repeated $k$-mers can introduce load imbalance amongst cores that further causes the run time to scale sub-optimally.

Weak scaling experiments showed a slight increase of run time as $p$ increased for the `find` operation (Figure 2.10b) for the hashed count index. This is due primarily to the "query" step. As the per-processor data size $|M|/p$ is kept constant in weak scaling experiments, both the latency and bandwidth terms in the communication complexity increase with $\log(p)$.

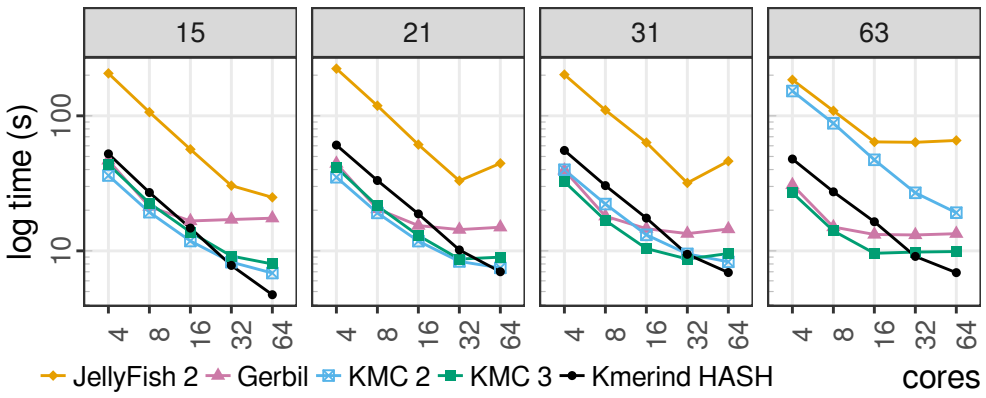
The `find` operation in a position index scales linearly with $p$ and $r$ according to $\tau p + (\mu+1)(r|M_i|)$ (Section 2.1.3). Assuming uniform sampling of a true $k$-mer distribution, $r$ is expected to increase with data set size, which scales with $p$ for weak scaling experiments. For data sets **M1–M5**, $r$ values are $1.11$, $1.17$, $1.26$, $1.37$, and $1.54$ respectively. Figure 2.10b illustrates this linear scaling behavior in the "query" and "resp" steps of the position index `find` operation.

### 2.3.5 Comparisons with Existing Tools

We compared the performance of Kmerind hashed and sorted count indices to existing best-in-class $k$-mer counting tools on shared- and distributed- memory systems. JellyFish 2 [26] is the *de facto* standard $k$-mer counter. We also compared to several more recent, state-of-the-art tools including KMC 2 [31], its successor KMC 3 [33], and Gerbil [32]. Kmernator [34] is chosen as it is the only existing, stand-alone, distributed $k$-mer counter.

*Shared-Memory Environment*

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




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

Figure 2.11: Strong scaling behavior for counting DNA $k$-mers in data set **R1**. Each plot shows the times in seconds to count a fixed size $k$-mer (15, 21, 31, 63) on increasing number of cores $p$ (4, 8, 16, 32, 64) in a shared-memory system. The table shows the total and counting-only (underlined) times for either fixed $p$ or fixed $k$. For readability, Kmerind's sorted count index results are shown only in the table.

possible to minimize disk usage for intermediate results.

As disk subsystem configurations can vary drastically, we minimized the impact of file I/O by leveraging the operating system cache (Section 2.3.2), and erasing output immediately after each run due to an observed high overhead to overwrite files on `ext4` file systems. For all software packages, we report the total running time including file input and output. For Kmerind and JellyFish 2, we also report the *counting-only* times which excluded result writing. We expect Kmerind's primary application usage pattern to involve constructing and querying in-memory indices, thus the scalability and absolute performance of the counting-only times is of practical importance. Figure 2.11 shows the

scalability of each software for each $k$ value, while the embedded table shows the running times for fixed $p = 64$ and varying $k$, and fixed $k = 31$ and varying $p$.

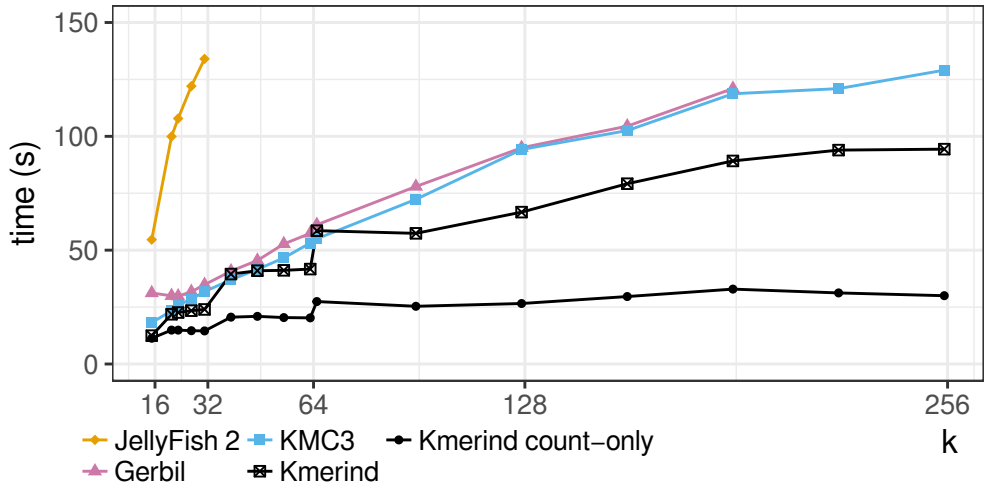



Figure 2.12: The time in seconds to count Canonical DNA $k$-mers for increasing $k$ (15, 21, 23, 27, 31, 39, 47, 55, 63, 65, 95, 127, 159, 191, 223, 255). JellyFish 2, KMC 3, Gerbil, and Kmerind's hashed count index were evaluated using data set **R2** on 64 cores. Gerbil failed for $k > 191$. JellyFish 2 times for $k$ larger than 31 exceeded 150 seconds and are excluded for readability.

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

Gerbil at low core counts.

The counting-only times of Kmerind's hashed and sorted count indices showed near linear scalability for up to 32 cores, beyond which the scalability decreased likely due to MPI communication complexity. The file writing time, as the difference between total and counting-only times, scaled sublinearly with $p$ and remained approximately constant at 2 seconds for 32 and 64 cores. These observations suggest that Kmerind's count index can continue to scale beyond 64 cores, but may be limited by file system performance when result writing is required. In contrast, Gerbil failed to scale beyond 16 cores for all $k$, while KMC 2, and KMC 3 had very limited scalability for $k \geq 31$ and $p \geq 16$, indicating that their performance bottlenecks may not be caused by file system limitations. JellyFish 2's counting-only times similarly suggest this hypothesis.

The dependence of KMC 2 on $k$ value was particularly pronounced for $k = 63$, where the counting time increased dramatically for all core counts tested. Kmerind, on the other hand, showed relatively constant running time for $k$ values of 21, 31, and 63, and a lower running time for $k = 15$. This behavior is attributable to a balance between the widening of $k$-mer representation from 32 bit to 128 bit, and the reduction in total $k$-mers in short-read data sets (Section 2.3.3) with increasing $k$. For the **R1** data set, the numbers of valid $k$-mers were 2052-, 1909-, 1670-, and 906-million for $k$ values of 15, 21, 31, and 63, respectively. Gerbil's reduction in running time is likely due to a similar cause, whereas KMC 2 and JellyFish 2's performance degradations suggest inefficiencies in $k$-mer parsing and comparison operations. We also note that while KMC 3 demonstrated significant improvement over KMC 2 for $k = 63$, for low $k$ KMC 2 actually performed better for most $p$ values.

We further evaluated the effects of varying $k$ up to 255 using the **R2** data set on 64 cores, shown in Figure 2.12. The values were chosen with consideration of C++ primitive type sizes. As KMC 3 was reported to significantly improve upon KMC 2's performance for high $k$ values [33], we included KMC 3 only.

Figure 2.12 further illustrates Kmerind's low sensitivity to increased in $k$. Kmerind

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

Table 2.6: Time in seconds to count canonical DNA 31-mers using 64 CPU cores. Underlined values represent the "counting-only" times.

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

Table 2.6 shows the performance of JellyFish 2, KMC 2, KMC 3, Gerbil, and Kmerind's hashed and sorted count indices for data sets of different sizes. All experiments used 64 cores to count canonical DNA 31-mers. The experiments with the metagenomic data sets demonstrated that all tools except for JellyFish 2 scaled nearly linearly with data size. KMC 2 was marginally faster than KMC 3 for all data sets except for **R3**, while Gerbil's performance lagged behind KMC 2 and KMC 3. Gerbil's performance for data set **G1** was unexpectedly but repeatably poor. Kmerind's sorted index performed comparably to KMC 3 for the metagenomic data sets and the assembled genomes, while the hashed count index outperformed all existing tools for all tested data sets. Kmerind's hashed index counted the **M3** data set in approximately $63.0$ seconds, the **R4** data set in $270.5$ seconds, the assembled human genome (G1) in $21.0$ seconds, and the pine genome (G2) in $70.6$

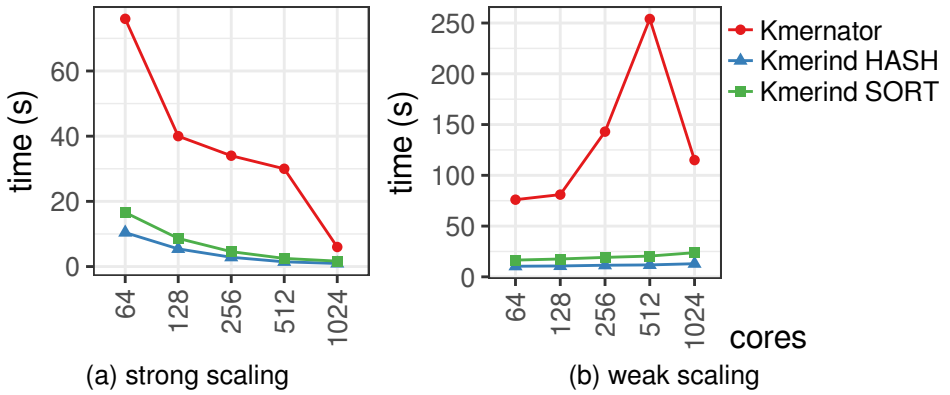
seconds. The hashed count index was therefore between $1.3\times$ and $1.6\times$ faster than KMC 2 and KMC 3.

Of the five existing tools tested, only JellyFish 2 includes a command line interface to query the index. KMC 2 and KMC 3 provide an option to find intersection between two indices, but they output the minimum counts for the entries in the intersection. We queried JellyFish 2 and Kmerind hashed count indices using $1\%$ of indexed $k$-mers on a single core, as JellyFish 2 supports single-threaded queries only. JellyFish 2 completed the query in $0.82$ seconds while Kmerind was able to do so in $0.11$ seconds. The results are not directly comparable as JellyFish 2 requires loading the database file from disk, but they are illustrative of the benefit of Kmerind's in-memory index for on-line queries. This point is further illustrated by Table 2.6, where the counting-only times of Kmerind's hashed count index is over $3\times$ faster than KMC 2 and KMC 3 for the metagenomic and assembled genome data sets, and approximately $2\times$ for the read sets **R2**, **R3**, and **R4**.

*Distributed-Memory Environment*

While Kmerind performs well in shared memory environments with high core counts, its performance advantages are further extended in a distributed memory environment. We benchmarked index construction for Kmerind's hashed and sorted indices and Kmernator using data sets **M1–M5** and 64 to 1024 cores on CyEnce.

Figure 2.13 shows both of Kmerind's indices were consistently faster than Kmernator by at least a factor of $6\times$ for strong scaling and $8\times$ for weak scaling. Kmerind's hashed count index completed 31-mer counting for the **M1** data set in $1.0$ seconds using 1024 cores, and Kmerind's hashed index showed approximately linear strong scaling for up to 512 cores, beyond which the parallel efficiency decreased slightly. For weak scaling, Kmerind's hashed index showed a gradual increase of running time as core count increased. In both cases, the behavior is attributable to the $\log(p)$ factor in the collective all-to-all communication complexity. Kmerind's sorted index was also consistently faster




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

Figure 2.13: Running times in seconds of Kmerind's hashed and sorted indices and Kmernator for counting canonical DNA 31-mer in strong and weak scaling distributed memory settings. Data set **M1** was used for strong scaling while sets **M1–M5** were used for weak scaling.

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

## 2.4 Summary

In this chapter we introduced Kmerind, the first distributed memory $k$-mer indexing library. It is a generic, template based library with sequential semantics and efficient parallel implementations. We described its API design philosophy and the operations supported.

We presented the bulk synchronous parallel algorithms behind the `insert`, `find`, `count` and `erase` operation implementations, including uni-index with hash table and sorted array back-ends, and algorithmic and implementation adaptation for multi-index.

We also presented our SIMD vectorization approach for the $k$-mer reverse complement operation, and the associated alphabet encoding modification that enabled vectorization for DNA 5 $k$-mers.

We characterized the performance of Kmerind while varying parameters including $k$ and the alphabet. Kmerind's performance and scaling were then demonstrated in the distributed memory and shared memory environments. We compared its speed to the current state-of-the-art $k$-mer counting tools and showed that it performed equally well or better at moderate to high core counts in shared memory environments, and it consistently out-performed these tools for high $k$ values.

In distributed memory, we showed that it out-performed the only existing distributed memory $k$-mer counter. It demonstrated good scaling behavior for experiments involving up to $1024$ cores, counting a 120GB metagenomic data set in less than $13$ seconds on 1024 Xeon CPU cores, and fully index the $k$-mer positions in $17$ seconds. Querying for $1\%$ of the $k$-mers in these indices can be completed in 0.23 seconds and 28 seconds, respectively.

# CHAPTER 3

# ARCHITECTURE AWARE OPTIMIZATIONS FOR K-MER COUNTING

The performance of Kmerind depend not only on efficient algorithms, but also on efficient implementations that leverages the system hardware features as appropriate and communication patterns. While we have shown that Kmerind performs well compared to other contemporary state-of-the-art software tools, there existed ample opportunities to improve its performance.

In this chapter, we explore architecture aware approaches to improve performance of Kmerind , including the use of SIMD hardware instructions, cache friendly hash table algorithm, software prefetching, and tighter integration between hash table operations and MPI communications.

The performance of a distributed hash table depends on the efficiencies of its communication algorithm and implementation and the operations of the local, sequential hash table. The performance of the local hash table in turn depends in large part on how fast it can hash and compare keys, and access the in memory data structure.

We optimized the distributed hash table implementation underlying the Kmerind library, with $k$-mer counting as its primary application use case. Distributed memory $k$-mer counting has the characteristics that $(1)$ many small $k$-mers, distributed across many cores and nodes, are $(2)$ communicated across the network, and stored in or queried against local data structures that tend to be $(3)$ memory latency- rather than compute-bound.

We address each of the characteristics, and make significant improvements to $(1)$ the MurmurHash3 hash function performance for batch-hashing small keys, such as $k$-mers, through vectorization, $(2)$ distributed hash table scaling behavior and communication overheads, and $(3)$ sequential hash table performances through cache-friendly algorithm designs that reduces memory access latency and bandwidth utilization.

Applications and libraries that utilizes distributed hash tables such as our de Bruijn graph library, Bruno, will benefit from the performance improvements presented here, provided they have similar characteristics as outlined above.

### 3.1 Preliminaries

We deviate slightly from the notations presented in Chapter 2 as a significant part of the discussion focuses on sequential hash tables.

A $k$-mer $\gamma$ is defined as a length $k$ sequence of characters drawn from the alphabet $\Sigma$. The space of all possible $k$-mers is then defined as $\Gamma = \Sigma^k$. Given a biological sequence $\mathcal{S} = s[0 \ldots (n-1)]$ with characters drawn from $\Sigma$, the collection of all $k$-mers in $\mathcal{S}$ is denoted by $\mathcal{K} = \{s[j \ldots (j+k-1)], 0 \leq j < (n-k)\}$.

$K$-mer counting computes the number of occurrences $\omega_l$, or frequency, of each $k$-mer $\gamma_l \in \Gamma$, $0 \leq l < |\Sigma|^k$, in $\mathcal{S}$. The *$k$-mer frequency spectrum* $\mathcal{F}$ can then be viewed as a set of mapping $\mathcal{F} = \{f : \gamma_l \rightarrow \omega_l\}$.

$K$-mer counting then requires first transforming $\mathcal{S}$ to $\mathcal{K}$ and then reducing $\mathcal{K}$ to $\mathcal{F}$ in $\Gamma$ space. We note that $k$-mers in $\mathcal{K}$ are arranged in the same order as $\mathcal{S}$, while $\Gamma$ and $\mathcal{F}$ typically follow lexicographic ordering established by the alphabet. The difference in ordering of $\mathcal{K}$ and $\mathcal{F}$ necessitate data movement when computing $\mathcal{F}$, and thus communication in distributed environment.

We further note that the storage representation of $\mathcal{F}$ affects the efficiency of its traversal. While $\Gamma$ grows exponentially with $k$, genome sizes are limited and therefore $\mathcal{F}$ is sparse for typical $k$ values. Associative data structures such as hash tables are well suited, which typically support the minimal operations of construction (or *insert* and *update*) and *query* of the $k$-mer counts.

We focus on efficient parallelization of data movements associated with the $\mathcal{K}$ to $\mathcal{F}$ reduction in distributed memory environments, and on the design of efficient hash tables for constructing and querying $\mathcal{F}$.

## 3.2 Sequential Hash Tables

In this section we focus on optimizing in-bucket element access for sequential hash tables. We use the following notations. A hash table element is defined as a key-value pair $w = \langle k, v \rangle$. The table consists of $B$ buckets. The id of a bucket is denoted as $b$. The hash function associated with the hash table is $H(\cdot)$. For open addressing hash tables, the *ideal* bucket id for an element $w$ is $b_w = H(w.k)\%B$, to distinguish from the actual bucket position where the element is stored. The *probe distance* $r$ of an element is the number of hash table elements that must be examined before a match is found or a miss is declared. For open addressing with linear probing, the probe distance is $r = b - b_w$.

### 3.2.1 Hash Table Collision Handling Strategies

Three factors affect sequential hash table performance: hash function choice, irregular access to a hash table bucket, and element access within a bucket, namely search for matching key. Hash function performance depends strongly on implementation and hardware capability, while the common perception is that hash tables are inherently cache inefficient due to the irregular memory accesses rendering hardware prefetching ineffective.

Intra-bucket data access performance, on the other hand, is an inherent property of the hash table design, namely the collision handling strategy. Collision occurs when multiple *keys* are assigned to the same hash table bucket, necessitating multiple comparisons and memory accesses. High collision rate therefore causes a deviation from the amortized constant time complexity.

Hash table design and implementation has been studied extensively. Cormen *et. al.* [83] described two basic hash table designs, *chaining* and *open addressing*. Briefly, chaining uses linked list to store collided entries, while open addressing maintains a single array and employs deterministic *probing* logic to locate an alternative insertion location when collision occurs. Identical probing logic is used during query.

Different probing algorithms exist. Cormen *et. al.* presented *linear* probing, where the hash table is searched linearly for empty bucket for insertion. Linear probing is sensitive to non-uniform distribution of the input, sub-optimal hash functions, high load, and even insertion order, all of which affect hash table operation times. Strategies such as quadratic probing and double hashing [83], Cuckoo hashing [84], HopScotch hashing [85], and Robin Hood hashing [86] aim to address these shortcomings. In all these strategies save Robin Hood hashing, key-value tuples from different buckets are interleaved. Hash table operations thus also require comparisons with keys from other buckets, increasing the average number of probes and running time.

The choice of collision handling strategy can affect memory access patterns for in-bucket traversal. Double hashing, HopScotch, and quadratic probing introduce irregular memory accesses, while linear probing and Robin Hood require only sequential memory access, thus can benefit from hardware prefetching. They also maximize cache line utilization and reduce bandwidth requirement.

### 3.2.2 Robin Hood Hashing

**Algorithm 3.1** Robin Hood Hashing: `insert`

1: $H()$ : hash function; $\langle k, v \rangle$ : key-value pair; $P_{rh}$ : array $[0..B-1]$ of key-value pairs

2: $b \leftarrow H(k)$ modulo $B$
3: $p \leftarrow b$
4: **while** $(p < B)$ AND $P_{rh}[p]$ is not empty **do**
5: $\quad b' \leftarrow H(P_{rh}[p].k)$ modulo $B$
6: $\quad$ **if** $(p - b) > (p - b')$ **then**
7: $\quad\quad$ `swap`$(\langle k, v \rangle, P_{rh}[p])$
8: $\quad\quad b \leftarrow b'$
9: $\quad$ **else if** $(p - b) == (p - b')$ AND $(k == P_{rh}[p].k)$ **then return**
10: $\quad$ **end if**
11: $\quad p \leftarrow p + 1$
12: **end while**

13: $P_{rh}[p] \leftarrow \langle k, v \rangle$

Robin Hood Hashing is an open addressing strategy whose objective is to minimize the expected probe distance, and thus the insertion, and query times. It was first proposed by Pedro Celis in his doctoral thesis [86]. We briefly describe the basic algorithms here.

During insertion, (Algorithm 3.1), a new element $w$ swaps with element $u$ in position $b$ if $w$ has a high probe count than $u$, i.e. $(b - b_w) > (b - b_u)$. The insertion operation continues with the swapped element until an empty bucket is found, where the swapped element is then inserted. This strategy results in the spatial grouping and ordering of table elements by bucket ids, resembling sorting.

Query, or *find* operation in a Robin Hood hash table is similar algorithmically. Search completes when an entry is found, the bucket is empty or when $(b-b_w) > (b-b_u)$ indicates that the end of the target bucket has been reached, thus the element is not found.

Deletion in a Robin Hood hash table first performs a search and then removes the found element. Celis' original algorithm uses tombstones to mark deleted entries and describes an optimized algorithm for insertion after deletion. More recent implementations [87] uses *backward shift* to maintain spatial coherence of bucket elements. The elements between the deleted entry and the next empty position are shifted backward to fill the newly emptied position.

### 3.2.3 Optimized Robin Hood Hashing

The "classic" Robin Hood hashing strategy spatially groups elements of a bucket together. This reduces variance of $r$ and eliminates interleaving of buckets, thus reducing the number of elements that must be compared and minimizing memory bandwidth requirement. However, classic Robin Hood begins a search from the ideal bucket, $b_w$. The elements in the range $[b_w \ldots (b_w + r)]$ are expected to belong to buckets with ids $b < b_w$. On average, classic Robin Hood must access $r$ elements, compute their hash values, and compare probe distances, before arriving at the elements of the desire bucket. We can avoid the extraneous work by storing the distances to the first bucket elements.

*Data Structure*

Our extended Robin Hood hash table, $T_{rh}$, consists of the tuple $\langle P_{rh}[], I_{rh}[] \rangle$. $P_{rh}[]$ is the primary one dimensional array that stores the elements of the hash table. Each element in $I_{rh}[]$, $I_{rh}[b]$, consists of a tuple $\langle$ empty, offset $\rangle$ $>$. The *empty* field indicates whether bucket $b$ is empty, while the *offset* field contains the probe distance from position $b$ to the first element of bucket $b$, which is then located at $P_{rh}[b + I_{rh}[b]]$. In the case where $I_{rh}[b]$ is empty and $offset$ is not zero, $I_{rh}[b]$ references the insertion position for the first element of bucket $b$.

Storing the probe distances in the $I_{rh}[]$ array avoids $r$ hash function invocations and $P_{rh}[]$ memory accesses. Consequently computation time and memory bandwidth utilization are further reduced compared to classic Robin Hood. We assume that insertion occurs less frequently than queries and therefore optimizations that benefit query operations, such as the use of $I_{rh}[]$, are preferred over those for insertion. As $r$ is expected to be relatively small [86], a small data type, e.g. an 8 bit integer, can be chosen to minimize memory footprint for $I_{rh}[]$ and to increase the cache-resident fraction of $I_{rh}[]$. The sign bit of $I_{rh}[b]$ correspond to $empty$ while the remaining bits correspond to *offset*.

We note that the $I_{rh}[]$ array is essentially the FastForward array in Ankerl's Robin Hood Hashing implementation [88]. In that implementation, the FastForward array is coupled to the "InfoByte" array that stores the reprobe distances. Since both need to be updated during insertion and deletion for every bucket between between the ideal bucket and the next empty slot, the benefits of each is nullified while $2\times$ the memory must be used.

*Insert Operation*

The optimized Robin Hood hash table insertion algorithm is outlined in Algorithm3.2. The range of the bucket in $P_{rh}[]$ is first computed in constant time in Line 3. Elements in the bucket are then compared for match in Lines 4–8. If matched then the function terminates. If a match is not found within the bucket, then the input element is inserted into $P_{rh}[]$ in the

**Algorithm 3.2** Optimized Robin Hood Hashing: `insert`

1: $H()$ : hash function; $\langle k, v \rangle$ : key-value pair to insert; $T_{rh} : \langle P_{rh}, I_{rh} \rangle$

2: $b \leftarrow H(k)$ modulo $B$
3: $p \leftarrow b + I_{rh}[b]$.offset; $p1 \leftarrow b + 1 + I_{rh}[b+1]$.offset
4: **while** $p < p1$ **do**
5: **if** $(P_{rh}[p].k == k)$ **then return**
6: **end if**
7: $p \leftarrow p + 1$
8: **end while**
9: $p \leftarrow b + I_{rh}[b]$.offset
10: **while** $p < B$ AND NOT ($I_{rh}[p]$ is empty AND $I_{rh}[p]$.offset $== 0$) **do**
11: `swap`($\langle k, v \rangle$, $P_{rh}[p]$)
12: $p \leftarrow p + 1$
13: **end while**
14: $P_{rh}[p] \leftarrow \langle k, v \rangle$; $I_{rh}[b]$.empty $\leftarrow FALSE$
15: **while** $b < p$ **do**
16: $I_{rh}[b+1]$.offset $\leftarrow I_{rh}[b+1]$.offset $+1$
17: $b \leftarrow b + 1$
18: **end while**

same manner as in classic Robin Hood Hashing, i.e. via iterative swapping. Since elements from bucket $b + 1$ to $p$, where $p$ after Line 13 references the last bucket to be modified, are shifted forward, and their offsets $I_{rh}[(b + 1) \ldots p]$ incremented (Lines 15–18).

The number of $P_{rh}[]$ elements to shift and $I_{rh}[]$ elements to update can be significant, up to $O(\log(B))$ according to Celis. We limit this shift distance indirectly by using only 7 bits for the offsets in $I_{rh}[]$. Overflow of any $I_{rh}[]$ element during insertion causes the hash table to resize automatically.

The small number of bits used for offset values and the contiguous memory access during the insert means that most of the updates can benefit from hardware prefetching and few cache lines need to be loaded.

*Find Operation*

Query operations in the optimized Robin Hood hash table follows closely the first part of the Insert algorithm (Algorithm 3.2 Lines 1–8).

*Erase Operation*

**Algorithm 3.3** Optimized Robin Hood Hashing: `erase`

```
 1: H() : hash function
 2: k : key to erase
 3: P_rh : array [0..B − 1] of key-value pairs
 4: I_rh : array [0..B − 1] of offsets

 5: b ← H(k) modulo B
 6: p ← b + I_rh[b].offset
 7: p1 ← b + 1 + I_rh[b + 1].offset

 8: while p < p1 do
 9:     if (P_rh[p].k == k) then
10:         break
11:     end if
12:     p ← p + 1
13: end while
14: if p == p1 then return
15: end if

16: while p < B AND ( I_rh[p].offset > 0) do
17:     P_rh[p] ← P_rh[p + 1]
18:     p ← p + 1
19: end while

20: b ← b + 1
21: while b < p do
22:     I_rh[b].offset ← I_rh[b].offset − 1
23: end while
24: I_rh[p].empty ← TRUE
```

The erase algorithm in the optimized Robin Hood hash table employs the backward-shift approach [87], namely that elements following the deleted entry are shifted backward (Lines 16–19) to remove gaps in a bucket's element range. We note that the offset array range $I_{hr}[(b+1)\ldots p]$ must be updated to reflect the shifted bucket start positions. Algorithm 3.3 illustrates the algorithm for erasing a single element.

We further optimize the erase operation when used in batch mode. Rather than immediately backward shift, we mark the element as deleted in a temporary array. After all

input keys have been processed, we backward shift the remaining elements to compact the buckets in $P_{hr}[]$, and recompute the offset array $I_{hr}[]$.

*Resize Operation*

**Algorithm 3.4** Optimized Robin Hood Hashing: `up-size`

```
 1: H() : hash function
 2: P_rh : array [0..B − 1] of key-value pairs
 3: I_rh : array [0..B − 1] of offsets
                                                    ▷ initialize new storage
 4: newP_rh ← array [0..2B − 1] of key-value pairs
 5: newI_rh ← array [0..2B − 1] of offsets
                                                    ▷ count entries in each new bucket
 6: for b ← 0 . . . (B − 1) do
 7:     if NOT I_rh[b].empty then
 8:         b′ ← H(P_rh[b].k) modulo 2B
 9:         newI_rh[b′].offset ← newI_rh[b′].offset+1
10:     end if
11: end for
                                                    ▷ transform counts into position offsets for each bucket
12: count ← newI_rh[0]
13: offset ← 0
14: for b′ ← 1 . . . (2B − 1) do
15:     newI_rh[b′ − 1].offset ← offset          ▷ offset values allow for previous empty buckets
16:     offset ← max(offset + count - 1, 0)
17:     count ← newI_rh[b′].offset
18: end for
                                                    ▷ copy elements from old storage to new
19: for b ← 0 . . . (B − 1) do
20:     if NOT I_rh[b].empty then
21:         b′ ← H(P_rh[b].k) modulo 2B
22:         newP_rh[b′ + newI_rh[b′].offset] ← P_rh[b]
23:         newI_rh[b′].offset ← newI_rh[b′].offset +1
24:         newI_rh[b′].empty ← FALSE
25:     end if
26: end for
                                                    ▷ shift offsets back by 1 position to get original offset.
27: offset ← 0
28: for b′ ← 0 . . . (2B − 1) do
29:     swap (offset,empty) with newI_rh[b′].(offset, empty)
30: end for
31: P_rh ← newP_rh
32: I_rh ← newI_rh
```

**Algorithm 3.5** Optimized Robin Hood Hashing: `down-size`

```
1: H() : hash function
2: P_rh : array [0..B − 1] of key-value pairs
3: I_rh : array [0..B − 1] of offsets
                                                    ▷ initialize new storage
4: newP_rh ← array [0..(B/2 − 1)] of key-value pairs
5: newI_rh ← array [0..(B/2 − 1)] of offsets
                                    ▷ generate the counts for each new bucket
6: for b' ← 0 ... (B/2 − 1) do
7:     count ← 0
8:     if NOT I_rh[b'].empty then
9:         count ← count +I_rh[b' + 1].offset +1 − I_rh[b'].offset
10:    end if
11:    if NOT I_rh[b' + (B/2)].empty then
12:        count ← count +I_rh[b' + (B/2) + 1].offset +1 − I_rh[b' + (B/2)].offset
13:    end ifnewI_rh[b'].offset ← count
14: end for
                                    ▷ compute offsets from the bucket counts
15: count ← newI_rh[0]
16: offset ← 0
17: for b' ← 1 ... (B/2 − 1) do
18:    newI_rh[b' − 1].offset ← offset          ▷ offsets allow previous empty buckets
19:    offset ← max(offset + count - 1, 0)
20:    count ← newI_rh[b'].offset
21: end for
                                    ▷ copy the elements to new position
22: for b ← 0 ... (B/2 − 1) do
23:    if NOT I_rh[b].empty then
24:        b1 ← b + 1
25:        newP_rh[b + newI_rh[b']] ← P_rh[b]
26:        newI_rh[b'].offset ← newI_rh[b'].offset +1
27:        newI_rh[b'].empty ← FALSE
28:    end if
29:    if NOT I_rh[b].empty then
30:        b1 ← b + 1
31:        newP_rh[b + newI_rh[b']] ← P_rh[b]
32:        newI_rh[b'].offset ← newI_rh[b'].offset +1
33:        newI_rh[b'].empty ← FALSE
34:    end if
35: end for
                    ▷ shift offsets back by 1 position to get original offset.
36: offset ← 0
37: for b' ← 0 ... (B/2 − 1) do
38:    swap (offset,empty) with newI_rh[b'].(offset, empty)
39: end for
```

Hash table resize is an expensive operation that requires first allocating a new table, potentially double in size, and then reinserting all existing elements in the table. As resizing typically occurs when the table is nearly full, the number of elements to resize is also at its maximum.

We have developed two algorithms for Robin Hood hash table resize, one for increasing the size of the hash table, and one for decreasing the size. We maintain power-of-2 size tables and therefore always up-size and down-size by a power-of-2. This design choice simplifies mapping between old and new buckets, and in the downsizing case, allows us to avoid hashing completely.

We observe that when the table doubles in size, an element in bucket $b_i$ is assigned to either bucket $b'_i = b_i$ or $b'_{i+B} = b_i + B$, where the bucket ids differ at the $\log(B)$ bit only. The relative order of the elements are maintained within the first and second $B$ buckets of the resized hash table. The arrays in the original table, and the first and second $B$ buckets of the new hash table, can be traversed linearly and concurrently. The insertion order is then in increasing bucket order thus avoiding the forward shift in the insertion algorithm. In addition, the linear traversal encourages hardware prefetching.

Similarly, halving the table size merges elements from buckets $b_i$ and $b_{i+B/2}$, $i < B/2$, to the new bucket $b'_i = b_i$ (Algorithm 3.5). In this case, it is not necessary to recompute the hash values, and the algorithm also avoids forward shifts in the insertion algorithm, and encourage hardware prefetching by linear traversal. Both algorithms readily generalize to resizing by higher powers of 2.

### 3.3 Distributed Hash Tables

We represent $\mathcal{F}$ as a distributed memory hash table. This allows us to encapsulate the parallel algorithm details behind the semantically simple `insert` and `find` operation interfaces following the general approach of Kmerind.

### 3.3.1 Kmerind Operations

In Chapter 2 we described Kmerind's design as utilizing a two-level distributed hash table. The first level maps $k$-mers to MPI processes using a hash function, while the second level exists as local hash tables. Different hash functions are used at the two levels to avoid correlated hash values which increases collision in the local hash table. The distributed hash table evenly partitions $\Gamma$ and therefore $\mathcal{F}$ across $P$ processes. We assume that $\mathcal{K}$ as the input is evenly partitioned across $P$ processes.

**Algorithm 3.6** Distributed Hash Table `insert`

1: $H()$ : hash function; $P$ : number of processes
2: $T$ : sequential hash table; $I[]$ : Input key-value array

3: $M[] \leftarrow H(I[])$ modulo $P$
4: $I'[] \leftarrow$ radixSort $I[]$ by rank $M[]$
5: $J[] \leftarrow$ distribute($I'[]$)
6: insert $J[]$ into $T$

The distributed hash table insertion algorithm proceeds in 3 steps, shown in Algorithm 3.6. Query operation follows the same steps, except an additional communication returns the query results to the source processes.

1. *permute*: The $k$-mers are assigned to processes via the top level hash function, and then rearranged via radix sort based on the process assignment. The input is traversed linearly, but the output array is randomly accessed. The first half of this step is bandwidth and compute bound while the radix sort is latency bound.

2. *alltoallv*: The rearranged $k$-mers are communicated to the remote processes via MPI_Alltoallv personalized collective communication.

3. *local compute*: The received $k$-mers are inserted into the local hash table.

**Algorithm 3.7** MPI_Alltoallv core communication algorithm

1: $I[][]$ : Input key-value array, grouped by process rank
2: $O[][]$ : Output key-value array, grouped by process rank
3: $P$ : number of processes
4: $r$ : rank of this process

5: **for** $i \leftarrow 0 \ldots (P-1)$ **do**
6: non-blocking recv $O[(r+i)$ modulo $P][]$ from rank $(r+i)$ modulo $P$
7: non-blocking send $I[(r+P-i)$ modulo $P][]$ to rank $(r+P-i)$ modulo $P$
8: **end for**
9: wait for all communication to complete

### 3.3.2 Communication Optimization

A typical MPI_Alltoallv implementation internally uses point-to-point communications over $P$ iterations to distribute data from one rank to all others, as shown in Algorithm 3.7. Its complexity is $O(\tau P + \mu N/P)$ where $\tau$ and $\mu$ are latency and bandwidth coefficients, respectively, and $N/P$ is the average message size for a processor. As $P$ increases, the latency term begins to dominate and scaling efficiency decreases. To manage the growth of communication latency, we sought to overlap communication and computation, and to reduce $P$ with a hybrid multi-node and multi-thread distributed hash table.

*Overlapped communication and computation*

**Algorithm 3.8** Alltoallv with overlapping computation

1: $I[][]$ : Input key-value array, grouped by process rank; $O[2][]$ : Output buffer
2: $P$ : number of processes; $r$ : rank of this process; $C()$ : computation to perform
3: **for** $i \leftarrow 1 \ldots (P-1)$ **do**
4: non-blocking send $I[(r+P-i)$ modulo $P][]$ to rank $(r+P-i)$ modulo $P$
5: **end for**
6: **for** $i \leftarrow 1 \ldots (P-1)$ **do**
7: non-blocking recv $O[(r+i)$ modulo $2][]$ from rank $(r+i)$ modulo $P$
8: $C(O[(r+i-1)$ modulo $2][])$
9: wait for non-blocking recv from rank $(r+i)$ modulo $P$ to complete
10: **end for**
11: $C(O[(r+i-1)$ modulo $2][])$
12: wait for all non-blocking sends to complete

We use overlapped communication and computation to hide some or all of the communication cost. Rather than waiting for all the iterations of point-to-point communication to complete, we allow computation to begin as soon as a point-to-point communication completes (Algorithm 3.8). This approach also allows buffers to be reused across communication iterations, thus reducing memory requirements.

Query operations can similarly leverage overlapped communication and computation. Once the computation is complete, a non-blocking send is issued, for which the matching non-blocking receives can be posted in Lines 3–5.

*Hybrid multi-node and multi-thread*

With large $P$, the latency term dominates in the communication time complexity. Small perturbation during execution due to communication overhead, load imbalance, or other system noise, can potentially propagate and amplify through all ranks in MPI_Alltoallv and our overlapped version (Algorithm 3.8).

To ameliorate this sensitivity, we reduce $P$ by assigning one MPI process per socket, and for each process spawn as many threads as there are cores per socket. Each thread instantiates its own local sequential hash table, and we partition $\Gamma$ across all threads on all processes. This approach maintains independence between local hash tables, thus inter-thread synchronization is minimized. We enabled multi-threading for the *permute* and *local compute* steps in the distributed hash table insertion algorithm, and the computation steps (Lines 8 and 11) in Algorithm 3.8.

## 3.4 Implementation Level Optimizations

We have implemented the algorithm described in Sections 3.3 and 3.2 using C++ and MPI primitives. Where appropriate, they are packaged as composable modules for Kmerind, such that the functionalities can be composed as needed by the application. We leveraged C++ 11 features as well as external libraries, including Google Dense Hash Map, Google

Farm Hash, and MurmurHash3 Hash.

The $k$-mer counting application, and likely other big data analytic problems, are often amenable to batch mode processing. We leverage this fact to enable vectorized hash value computation, cardinality estimation, and software prefetching for irregular memory access in the hash table.

### 3.4.1 Power of 2 bucket sizes

For Robinhood hash tables, we choose $B$ and $q_{rs}$ as powers of 2. Size and bucket id related computations can then be computed using single-cycle bitwise *shift* and *and* operations rather than expensive *division* and *modulo* operations, which can use up to 95 cycles for 64 bit integers on Haswell, Broadwell, and Skylake CPUs [89]. Since these computations often occur inside inner loops, the effects can be substantial.

### 3.4.2 Hash Function Vectorization

The choice of hash functions can have significant impacts for sequential and distributed memory hash table performances. Uniform distribution of hash values improves load balance for distribute hash tables and thus the communication and parallel computation time, and reduces collision rate thus sequential hash table performance. Computational performance of the hash function itself is also important.

Cormen *et. al.* qualified "good" hash functions as those that satisfy the *simple uniform hashing* assumption, that a key maps to a hash table bucket with probability $1/B$, where $B$ is the number of hash table buckets. For sequential hash tables, uniform hashing minimizes the collision rate, thus the time for insertion and query. In distributed hash tables, uniform hashing minimizes the load imbalance between processors, thus the communication and computation times. Simultaneously, absolute performance of the hash function impacts the overall hash table operation performance as they are applied potentially multiple times for each element.

For $k$-mer counting, the keys to be processed are numerous and limited in size, rarely exceeding $64$ bytes in length and often can fit in $8$ to $16$ bytes. Well known and well behaving hash functions such as MurmurHash3 and Google Farm Hash are designed for hashing single long key efficiently, however. Integer hashing, on the other hand, does not work well for larger $k$-mers.

We manually vectorized MurmurHash3's x$86$ variants of $32$- and $128$-bit hash functions using AVX and AVX2 SIMD instruction set. MurmurHash3, and in particular the x$86$ versions were chosen for their algorithmic simplicity and lower data dependencies. Farm hash extensively utilizes SIMD instructions and therefore is too complex to re-vectorize for short keys. Our vectorized MurmurHash3 hash functions computes $8$ $32$- or $128$-bit hash values concurrently. All operations in our sequential and distributed hash tables batch-hash the $k$-mers when the vectorized MurmurHash3 hash functions are chosen.

For a local hash table, where $B$ is less likely to exceed $2^{32}$, we also implemented a CRC32 based hash function, directly invoking the built-in CRC32C hardware instruction of the CPU. Transformation of this hash function into a family of hash functions would require significant algorithm engineering. We note that while CRC32C shows reasonable performance for our hash tables, its hash value distribution has not been rigorously evaluated. Consequently, we limit our use of the CRC32C hash function to only once in the distributed hash table hierarchy in order to avoid clumping of the elements.

### 3.4.3 Cardinality Estimation

Hash table resizing can be an expensive operation, as elements are reinserted into the expanded table. It is preferable to size the table as close to the final size as possible. We adopt cardinality estimation of unique $k$-mers for resizing the table ahead of batch insertion.

We implemented HyperLogLog++ [90] as its use of $64$-bit hash values supports larger genomes and $k$ values. We use batch mode vectorized MurmurHash3 hash function in HyperLogLog++ updates, and reuse the hash values during hash table insertion. Further-

more, since the estimator is invoked on data batch before insertion, any necessary hash table resizing occurs while the table contains fewer elements, thus resizing becomes faster as well.

We choose $12$ as the default HyperLogLog's precision value, corresponding to $m = 4096$ bins, which easily fit within L1 cache while still limit expected standard error to at most $1/\sqrt{m} \approx 1.56\%$. The number of bins also correspond to a medium cardinality threshold of $20480$. As a typical genome or read file contains millions of bases and similar number of $k$-mers, it is unlikely that the medium and lower range correction is required. At the upper end, large genomes, high $k$ value, and large sequencer output can easier produce over $2^{32}/30$ or $\approx 143$ million unique $k$-mers. HyperLogLog++, rather than the original 32-bit HyperLogLog [91] where the original 32-bit HyperLogLog algorithm requires a large value correction.

We implemented a modified HyperLogLog++ algorithm [90] that maintains the 64-bit hash compatibility while foregoing the medium cardinality range correction and the sparse representation. We use the same hash function for the hash tables and for HyperLogLog++, which allows us to reuse the computed hash values. The estimator has been integrated into Robinhood hash tables.

### 3.4.4 Software Prefetching

While individual hash table *insert* and *find* operations incur irregular memory access, with batch mode processing, future memory accesses can be computed thus software prefetching can be employed to reduce or hide memory access latencies.

For each operation, we compute the hash values and bin ids $b_w$ in batch. The software prefetching intrinsics are issued some iterations ahead of a bucket or bin's actual use. The number of iteration is referred to as the prefetch distance. In the Robinhood hashing scheme, the arrays $P_{rh}[]$ and $I_{rh}[]$ are prefetched via this approach. The *permute* step of distributed hash table operations likewise benefits from software prefetching.

Thus we began with a problem that suffers from poor data locality requiring large amounts of data to be read from memory, and formulated a new algorithm that improves the data locality and reduces the memory IO requirement and used software prefetching to convert the latency bound problem to bandwidth bound. We also vectorized hash table computation and picked parameter values to prevent any expensive arithmetic operations.

## 3.5 Performance Evaluations

Table 3.1 details the data sets used in our studies. We compare our optimized implementation to the Kmerind that uses Google Dense Hash Map, Farm Hash, and MurmurHash3 hash, and other existing tools for $k$-mer counting.

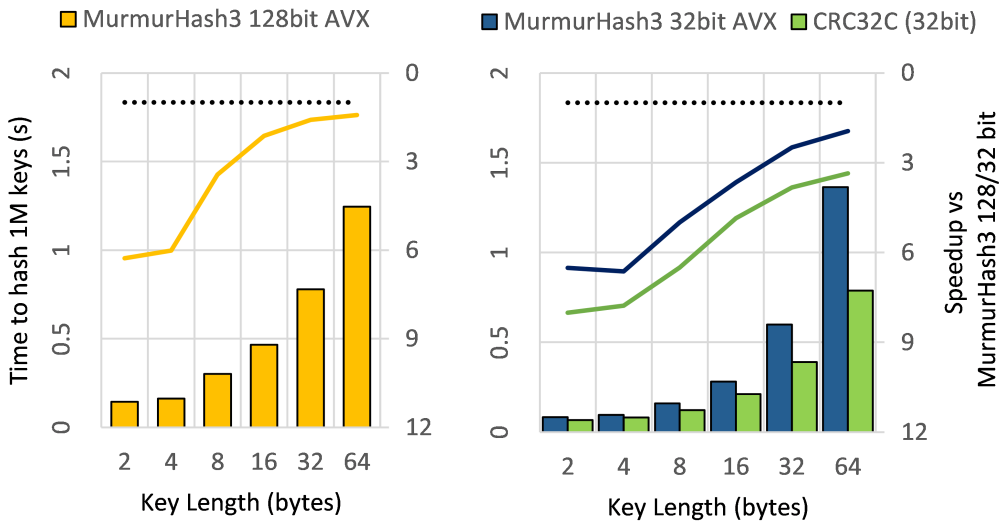
The shared-memory experiments, including comparison to existing tools, were conducted on CompBio, a single-node quad-socket Intel® Xeon® CPU E7-8870 v3 (Haswell) system with $18$ cores per socket running at $2.1$ GHz and with 1TB of DDR4 memory. All binaries were compiled using GCC 5.3.0 and OpenMPI 1.10.2. We conducted our multi-node experiments on the Cori supercomputer (Phase I partition). Each Phase I node has 128 GB of host memory and a dual-socket Intel® Xeon® CPU E5-2698 v3 (Haswell) with $16$ cores per socket running at 2.3 GHz. The nodes are connected with Cray Aries interconnect with Dragonfly topology with 5.625 TB/s global bandwidth. We use cray-MPICH 7.6.0 and ICC 18.0.0.

### 3.5.1 Hash Function Comparisons

We compare the performance of our vectorized implementations of MurmurHash3 32- and 128-bit hash functions with the corresponding scalar implementations for different key sizes in Figure 3.1 using a sequential benchmark code. The keys were randomly generated to simulate encoded DNA strings. As typical $k$ values for $k$-mers are less than 100, our hash function performance test used keys with power-of-2 lengths up to $64$ bytes covering a $k$ value of up to $128$ for DNA-IUPAC that requires encoding with $4$ bits per base. Our vector-

Table 3.1: Experimental data sets used for all evaluations. Where applicable, accession numbers for NCBI are provided.

| Id | Organism | File Count | File Size (GB) | Source | Accession/ Notes |
|---|---|---|---|---|---|
| R1 | H. sapiens | 1 | 6.3 | NCBI | SRR077487 |
| | | | | 1000 Genome HG00096 | forward reads only |
| R2 | F. vesca [81] | 11 | 14.1 | NCBI | SRA020125 |
| R3 | G. gallus | 12 | 115.9 | NCBI | SRA030220 |
| R4 | H. sapiens | 48 | 424.5 | NCBI | ERA015743 |
| R5 | H. sapiens | 6 | 18 | GAGE | Human Chr14 |
| R6 | B. impatiens | 8 | 151 | GAGE | Bumble Bee |
| R7 | H. sapiens | 1 (of 6) | 324 | NCBI | SRP003680 |
| G1 | H. sapiens | 1 | 2.9 | 1000 Genome GRCh37 reference assembly | assembled |
| G2 | P. abies [82] | 1 | 12.4 | Congenie.org | assembled |
| M1 | | 1 | 7.6 | IOWA Continuous | |
| M2 | metagenome | 1 | 15.2 | Corn Soil ( Project 402461 ), | |
| M3 | | 1 | 30.4 | Joint Genome Institute | |




(a) Hash to 128 bits

(b) Hash to 32 bits

Figure 3.1: Performance of SIMD vectorized MurmurHash3 hash functions, 3.1a for the 128 bit hash function, and 3.1b for the 32 bit hash function. The bars show time consumed by AVX vectorized hash functions and hardware assisted CRC32-based hashing. The lines show speedup of the AVX implementation over the scalar counterpart.

ized implementations show up to $6.5$x speedup over the respective scalar implementations. As key size increases towards cache line size and beyond, the overhead of reorganizing data across multiple cache lines for vectorization offsets any performance gains.

### 3.5.2 Hashing Schemes

*Hash table parameters*

The primary parameters for the hash table are the prefetch distance for memory access, and the table load factor. In our Robinhood schemes (referred to as RH henceforth), all hash table operations begin with accessing $I_{rh}[b]$ for bucket $b$, followed conditionally by comparing $P_{rh}[b + I_{rh}[b]]$ if the bucket is occupied. We chose the prefetch distances of $P_{rh}[]$ as $8$ empirically, and consequently set the prefetch distance for $I_{rh}[]$ at twice that.

The load factor is inversely proportional to memory requirement and throughput. We experimented with multiple values of the load factor between $0.5$ and $0.9$ and found that the throughput is nearly the same between values of $0.5$ and $0.8$ and reduces after that. Hence, we picked the value of $0.8$ to get nearly the best performance with minimum memory requirement.

*Sequential Comparison of various hashing schemes*

Figure 3.3 compares the sequential performance of our hashing schemes with the other prominent ones as table size grows. In order to isolate the comparison of only the hashing scheme and not the hash function performance, we fixed the hash function across all the hashing schemes. We picked MurmurHash3 128-bit as the hash function as source codes of some of the hashing schemes are not designed to use a vectorized hash function.

In the *insert* plot, the sudden jumps in time consumed are due to hash table being resized and correspond to the jumps in Figure 3.3c. For *insert*, RH bested RH Classic and Densehash for all iterations, with speedups of up to $3\times$ and $2.7\times$, respectively. For *find*, RH achieve speedups of up to nearly $4\times$ and $4.4\times$ over RH Classic and Densehash, and

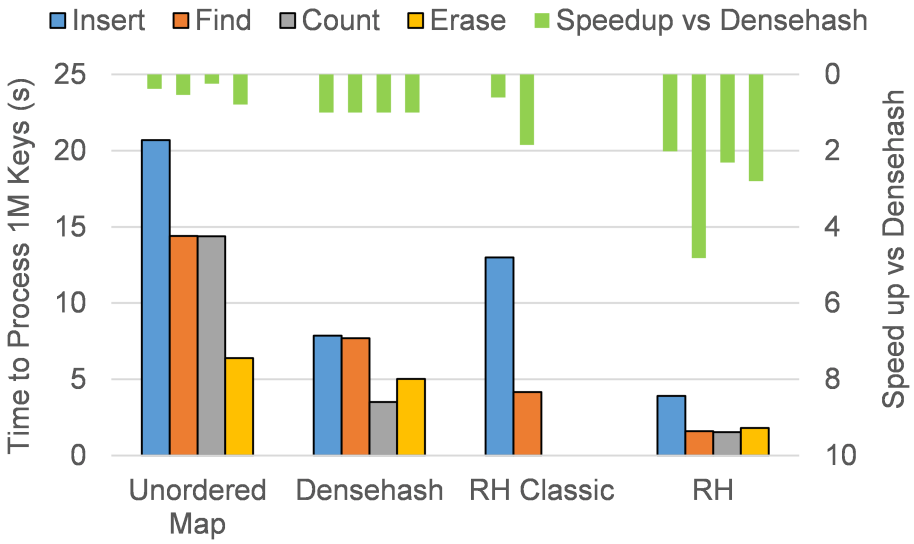
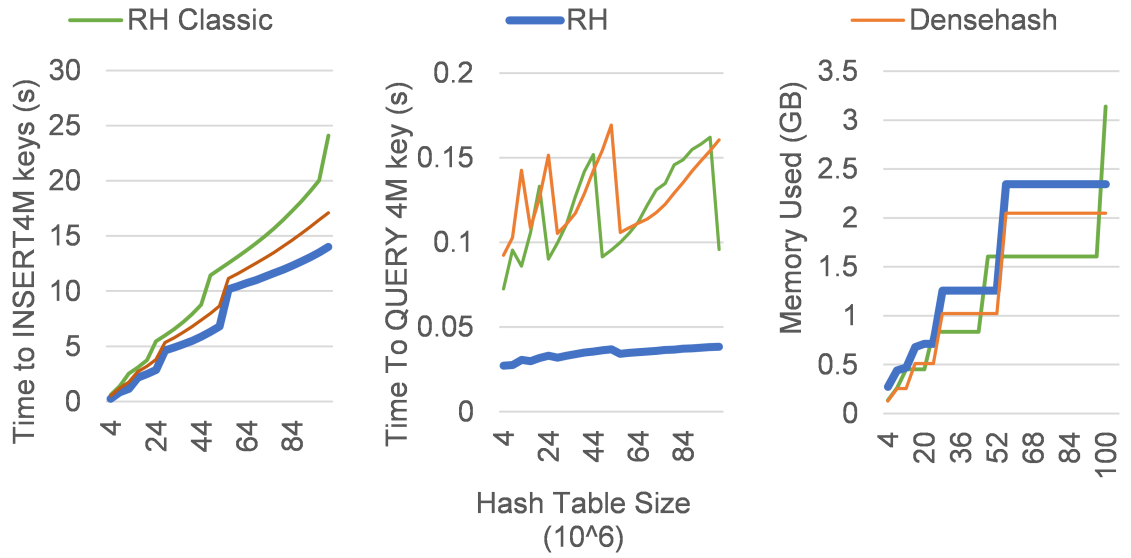



Figure 3.2: Comparison of sequential performance of different hashing schemes. Densehash refers to Google Dense Hash Map. RH Classic refers to the original Robin Hood hashing scheme, and the implementation is courtesy of Martin Ankerl [88]. Speed up vs Google Dense Hash Map for each hash table operation is shown as green bars with increasing value downwards.

the times consumed showed only small dependence on hash table size and load compared to RH Classic and Densehash. The memory consumption of RH, Google Dense Hash Map and RH classic are relatively closer to each other with RH classic needing the least amount in most cases.

### 3.5.3 Speedup with respect to optimizations on a single node

Figure 3.4 compares the performance of Kmerind (KI) with RH based hash tables as various optimizations are enabled from left to right. For RH hashing scheme, each bar shows the cumulative effect of all the optimizations up to that data point. The left-most bar represents the performance using scalar MurmurHash3 128-bit hash function for both *Permute* and *Local compute* stages and with *software prefetching* OFF. The next bar has *software prefetching* turned ON. The subsequent bar additionally uses vectorized version of MurmurHash3 128-bit hash function. The last bar additionally replaces vectorized MurmurHash3 128-bit hash function with CRC32C hash function for the *Local compute* stage.




(a) Insert 4 M keys (b) Find 4 M existing keys (c) Memory Usage

Figure 3.3: Comparison of sequential performance of different hashing schemes as the table size is increased incrementally by inserting randomly generated integer tuples $\langle$64-bit key, 32-bit value$\rangle$. Densehash refers to Google Dense Hash map and uses quadratic probing. RH Classic refers to the original Robinhood hashing scheme [86], and the implementation is from `https://github.com/martinus/robin-hood-hashing`. In each iteration, 4 Million keys are inserted and queried without clearing the hash tables.

The performance of *Permute* and *Local compute* stages improve with the use of *software prefetching* and *vectorization*. Moreover, *Local compute* stage gets further accelerated with the use of CRC32C hash function. Our best implementations for each operation achieve a speedup of 2.4-2.6x over the previous Kmerind implementation that uses Google Dense Hash Map.

### 3.5.4 Comparison to existing $k$-mer counters on a single node

Since majority of the existing $k$-mer counters are only built for shared memory systems, here we compare our performance with existing $k$-mer counters on a single node, mapping one MPI rank per core.

First, we compare the performance of these tools over a wide range of values of $k$ (Figure 3.5). Since file IO is an integral part of the $k$-mer counting algorithm of KMC3

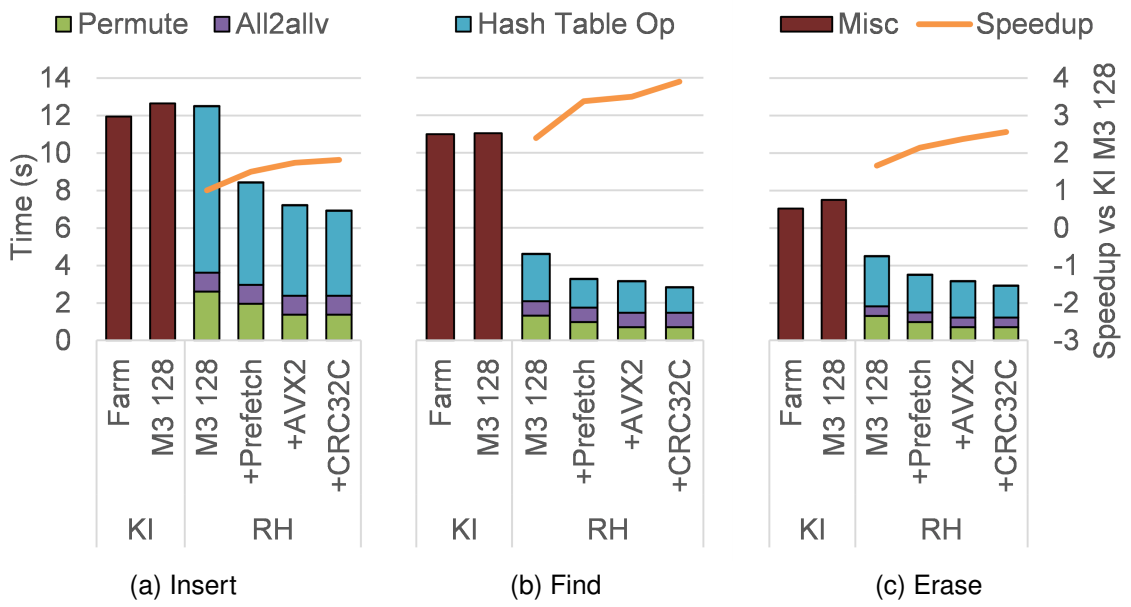



Figure 3.4: Effects of optimizations on hash table operation performance on a single node. M3 refers to MurmurHash3. The line plot shows speedup relative to the corresponding operations in Kmerind with M3 128. # MPI ranks: 64, data set: R5.

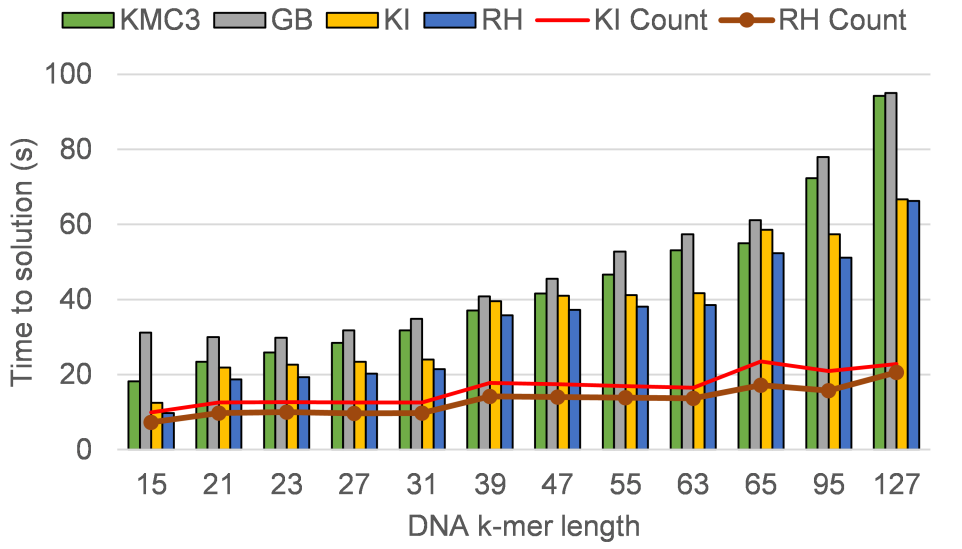



Figure 3.5: Effect of varying $k$ on total time to count $k$-mers in the F. vesca data set (R2). The line plots with markers represent the total time spent by KMC 3, Gerbil (GB), and Kmerind (KI). The bars indicate the total time used by RH The lines plots without markers represent the time spent during counting by KI and RH exclusive of file I/O times. KMC3 and GB were run on CompBio with 64 threads, while RH and KI were run using 64 MPI ranks.

and Gerbil, we compare all the tools on total time. We also compare RH and KI on time spent in just $k$-mer counting as these tools use file IO only to read input and write output. For all values of $k$, our implementations are faster than the other tools. On total time, we achieve up to 2x and 3.5x speedup over KMC3 and GB respectively. For the counting step only, our implementations achieve up to 1.6x speedup over KI.

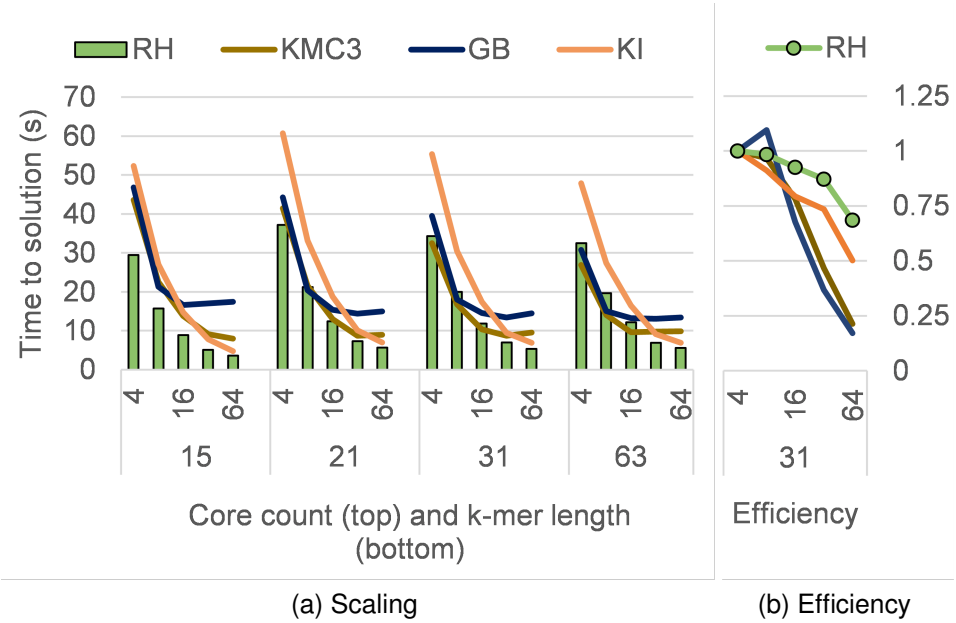



Figure 3.6: Comparison of strong scaling performance of KMC 3, GB, KI, and RH on a single node. $k$ is varied to show effects of changing key size. Parallel efficiencies are shown for $k = 31$ only as the efficiency behaviors for the $k$-mer counters are similar for different $k$ values. data set: R1

Figure 3.6 compares the strong scaling performance of various tools on total time inclusive of file IO. Since $k$-mer counting operation is throttled by file IO and memory, these two factors play a key role in scaling within a node. KMC3 and GB show poor scaling beyond 8 threads for GB and 16 threads for KMC3, achieving scaling efficiencies of $0.21$ and $0.17$ at $64$ cores respectively. This is attributable to heavy use of file IO by these tools and synchronization costs. KI and RH scale significantly better. The scaling of KI and RH appears low due to poor scaling efficiency of file IO ($0.22$-$0.4$ at $64$ cores). Note that optimizing file IO is beyond the scope of this paper.

Excluding file IO, RH and KI achieve scaling efficiencies of $0.56$ and $0.68$, respectively.

While KI is significantly slower than RH at $64$ cores, it achieves better scaling due to significantly worse single core performance from a memory latency bound implementation.

Table 3.2: Comparison of time consumed (in seconds) by various $k$-mer counters on real data sets. JF, KMC2, KMC3, and GB were run with $64$ threads, while KI and RH were run using $64$ MPI ranks. The "Counting Only" rows exclude file IO time.

| K | M1 | M2 | M3 | R2 | R3 | R4 | G1 | G2 |
|---|---|---|---|---|---|---|---|---|
| JF | 133.36 | 207.47 | 321.78 | 127.84 | 347.28 | 1465.86 | 132.65 | 329.45 |
| KMC2 | 22.84 | 41.89 | 82.59 | 29.24 | 122.11 | 432.92 | 30.03 | 100.35 |
| KMC3 | 24.20 | 45.16 | 86.47 | 31.80 | 99.41 | 455.96 | 31.03 | 102.15 |
| GB | 26.30 | 50.68 | 97.15 | 34.83 | 184.31 | 696.62 | 1235.82 | 153.09 |
| KI | 16.78 | 32.35 | 62.99 | 23.97 | 77.90 | 269.96 | 21.01 | 70.59 |
| RH | 15.74 | 30.59 | 57.55 | 21.13 | 66.34 | 239.59 | 18.17 | 61.68 |
| JF Count | 22.11 | 38.06 | 77.54 | 27.95 | 215.21 | 1201.46 | 14.74 | 63.93 |
| KI Count | 5.58 | 11.08 | 22.10 | 12.53 | 52.06 | 190.29 | 8.42 | 27.84 |
| RH Count | 4.67 | 9.21 | 18.24 | 9.67 | 40.58 | 160.69 | 5.73 | 22.12 |

Next, we compare the performance of these tools on a set of large scale data sets (Table 3.2). The results show that we obtained significantly better performance, achieving higher speedups over existing tools for data sets with higher repeat factors. Since we set $B$ to the expected number of unique $k$-mers, higher repeat factor ensures smaller values of $B$. This in turn means smaller hash tables and better data locality. For RH, since updating an existing entry is a lot faster than creating a new one, higher repeat factors imply better performance.

### 3.5.5 Multinode Scaling

We compare the strong scaling performance of KI with various RH in Figure 3.7. At lower core counts, the notable benefit of overlapping *Local compute* and communication is evident. While hybrid MPI-OpenMP version does not perform as well due to a significantly higher *Wait* time from large message size. At this stage, communication is network bandwidth bound and a single communicating thread per socket is not sufficient to saturate the network bandwidth and execution of non-communication part of MPI operations within a socket is done serially. On the other hand, the hybrid version performs well at higher core

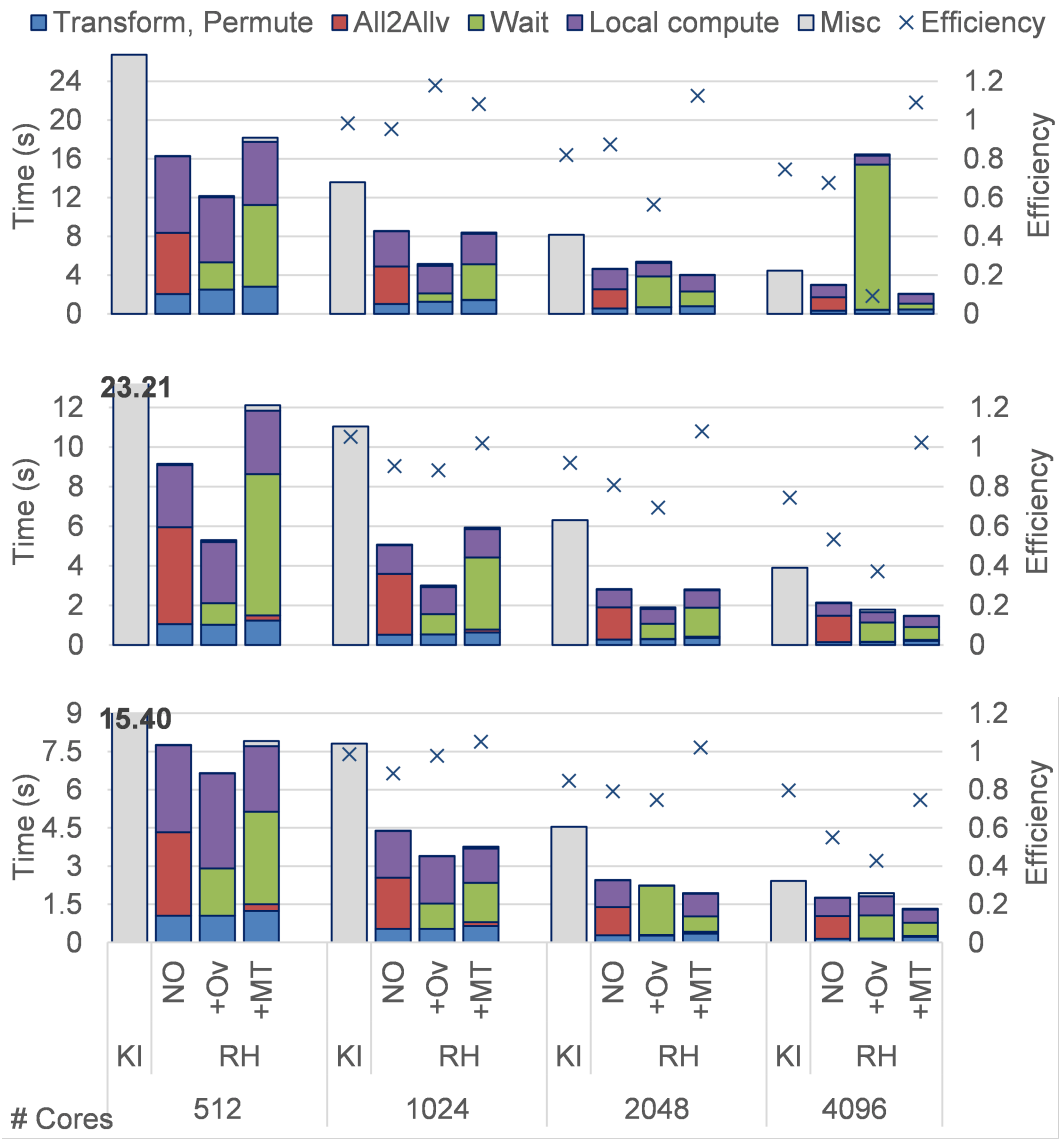



Figure 3.7: Performance improvements of optimized Kmerind in distributed memory environment. Strong scaling using 16 to 128 nodes of Cori with R6 data set. NO and Ov stand for non-overlapped and overlapped communication, respectively. For RH, the leftmost bar uses the best configuration from Figure 3 (with software prefetching, vectorized MurmurHash3 128-bit for *Permute* and CRC32C for *Local compute*). The next 2 bars additionally use overlapped compute and communication. The last bar additionally uses multi-threading. The markers show the scaling efficiency of each configuration compared to the run of the same configuration at 512 cores. For the +MT case, we use one MPI rank per socket and # cores/socket as # OpenMP threads/rank. For all other cases, # MPI ranks = # cores.

counts as the communication becomes network latency bound. In addition, in strong scaling setting, the data size $N/p$ decreases with increasing $p$, thus reducing the hash table size

and improving cache hit rate. As a result, the hybrid version achieves super-linear scaling in almost all cases. For the MPI-only version, *Wait* time is significantly higher at large core-counts due to reasons discussed in Section 3.3.

Overall, we achieve significantly higher performance compared to KI, up to $2.6$x and $4.4$x for *insert* and *find*, respectively.

We performed similar scaling experiments with the $324$ GB R7 human genome data set. We achieve speedup over KI of up to $2.44$x and $6.4$x for *insert* and *find*, respectively. *Insert* and *find* on 4096 cores completed in just $4.1$ and $6.7$ seconds respectively.

### 3.6 Summary

In this chapter we described some optimizations for the Kmerind library, targeting distributed $k$-mer counting.

We discussed our cache friendly batch-mode hash table that reduces memory access latency and bandwidth usage through an improved Robin Hood hashing scheme. We leveraged the batch-mode characteristics to enable software prefetching to further reduce memory access latency.

We then described our strategy to reduce communication overheads through overlapped communication and computation, and to reduce communication latency by decreasing the MPI process count through intra-socket multi-threading.

We described our general approach for SIMD vectorization of MurmurHash3 hash functions specifically for batch processing $k$-mers and other data types with short, fixed sized byte arrays.

Effects of such optimizations are significant. Our AVX2 vectorized MurmurHash3 hash functions achieved up to $6.5\times$ speedup for hashing small $k$-mers compared to the scalar MurmurHash3 implementation. Our batch-mode optimized Robin Hood hashing scheme out-performs Google Dense Hash Map by $2\times$ for the *insert* operations and $4.8\times$ for the *find* operation. The communication and computation overlap, coupled with hybrid

MPI-OpenMP implementation allowed effective scaling to $4096$ cores and completing the counting task for a $350$ GB data set in $4.1$ seconds.

# CHAPTER 4
# DE BRUIJN GRAPH

De Bruijn graphs based assembly is currently the *de facto* approach for *de novo* assembly of high throughput short sequencing reads. A high performance parallel de Bruijn graph data structure and the associated graph operations can significantly improve the speed of assembly and provide a common platform on which to compare diverse assembly algorithms.

Although existing de Brujin graph based assemblers differ in terms of graph representation and specific assembly approaches, conceptually they share the same assembly pipeline. Typically, the pipeline works in five major steps: ($i$) $k$-mer generation and de Bruijn graph construction, ($ii$) graph simplification (e.g. tip removal), ($iii$) graph compacting by compressing linear chains of unambiguously connected vertices, and ($iv$) contiguous sequences (*unitigs*) generation via path traversal, and ($v$) gap closing and scaffolding *unitigs* relying on the orientation and distance information from paired-end/mate-paired reads.

We present our work on a parallel de Bruijn graph library, **Bruno**, designed for distributed memory environments. The Bruno library is built on top of the distributed hash tables and indices from the Kmerind library as defined in Chapter 2, and inherits the Kmerind APIs, algorithm implementations, and bulk-synchronous model of computation and communication to minimize communication latency, while allowing MPI to optimize data movement. The following high level parallel operations are defined for the de Bruijn graph: (1) construction; (2) linear chain compaction; (3) pure cycle detection; (4) error detection and removal based on frequency and graph structures; and (5) foundational graph operations to support queries, filtering, and traversal.

In this chapter, we describe Bruno's support for the first three operations as well as describing the API design as related to (5). In Chapter 5, error detection and removal is

discussed.

We present an algorithm for constructing a de Bruijn graph from $k$-mers and their flanking characters, representing edges. A fast parallel algorithm for bi-directed de Bruijn graph chain compaction is then presented. Our algorithm compacts multiple chains in a de Bruijn graph concurrently and independently using provably logarithmic rounds of bulk synchronous communication. Our algorithm embodies the following contributions:

- The algorithm uses single-stranded paths and path canonicalization to account for and directly operate on a bi-directed de Bruijn graph without graph transformations, thus ensuring correctness and simplifying traversal logic.
- The algorithm compacts chains via bi-directional traversal, and uses a symmetric path representation to allow local pointer doubling during each iteration and reduce communication rounds.
- The algorithm simultaneously identifies cycle vertices during chain compaction without incurring additional space, time, and communication complexity costs. With a small constant overhead it can be extended to label cycle vertices.

### 4.1 Preliminaries

We explicitly model DNA molecules as double stranded. The sequence of nucleotides on a single strand of a DNA sequence of length $n$ is represented by a character sequence $s = s[0] \dots s[n-1]$, drawn from the alphabet $\Sigma = \{A, C, G, T\}$ and ordered along the $5'$ to $3'$ orientation of the strand. The complementary strand is represented by the *reverse complement* sequence $\overline{s} = c(s[n-1]) \dots c(s[0])$ where $c(\bullet)$ represents the character complement based on the mapping $A \leftrightarrow T$ and $C \leftrightarrow G$. A DNA molecule can be represented by the unordered tuple $\langle s, \overline{s} \rangle$, or succinctly by the *canonical* sequence $\hat{s}$, defined as the lexicographically smaller of $s$ and $\overline{s}$, $\hat{s} = \min(s, \overline{s})$. The lexicographically larger sequence is denoted as $\check{s}$ for convenience.

Table 4.1: Table of notations

| | meaning | example |
|---|---|---|
| $s$ | character sequence | $s =$ ATCGC |
| $\overline{s}$ | reverse complement of $s$ | $\overline{s} =$ GCGAT |
| $\langle s, \overline{s} \rangle$ | a DNA molecule | $\langle$ATCGC, GCGAT $\rangle$ |
| $m$ | a $k$-mer in $s$ | $m =$ ATC |
| $\hat{m}$ | canonical $k$-mer of $m$, $\hat{m} = min(m, \overline{m})$ | $\hat{m} =$ ATC |
| $\check{m}$ | non-canonical $k$-mer of $m$, $\hat{m} = max(m, \overline{m})$ | $\check{m} =$ GAT |
| $\langle \hat{m}, \check{m} \rangle$ | a $k$-molecule | $\langle$ATC, GAT $\rangle$ |
| ${}_i\ddot{m}_j$ | denotes the $k$-mer $m_j$ with the additional constraint that it is on the same DNA strand as $k$-mer $m_i$ | $m_i =$ ATC , ${}_i\ddot{m}_j =$ CGC |
| $v_i$ | a vertex in a de Bruijn graph, $v_i \equiv \langle \hat{m}_i, \check{m}_i \rangle \equiv \hat{m}_i$ | $v_i =$ ATC |
| $(v_i, v_j)$ | an edge (with $k$-1 overlap) in a de Bruijn graph between vertices $v_i, v_j$ in 5′-to-3′ order, $(v_i, v_j) = (m_i, {}_i\ddot{m}_j)$ | (ATC, TCG) |

Let a $k$-mer $m$ be a length $k$ substring of $s$. The definitions and properties for $s$ and its derivatives apply directly to a $k$-mer and its derivatives: reverse complement $\overline{m}$, canonical $k$-mer $\hat{m}$, and $k$-molecule $\langle m, \overline{m} \rangle \equiv \langle \hat{m}, \check{m} \rangle$. We define the set of distinct $k$-mers in a sequencing read set $R = \{s\}$ as $M^k$, and the set of distinct canonical $k$-mers as $\hat{M}^k$. We similarly define ($k$+1)-mer, its derivatives, and the sets $M^{k+1}$ and $\hat{M}^{k+1}$. A ($k$+1)-mer can be considered as the result of merging its $k$-mer prefix $m_p = m[0 \ldots (k-2)]$ and suffix $m_s = m[1 \ldots (k-1)]$, both in $M^k$, on the same DNA strand, and by definition have a length $k$-1 suffix-prefix overlap. Table 4.1 summarizes our notations.

### 4.1.1 Bi-directed de Bruijn Graph

A de Bruijn graph is a directed graph whose vertices are $k$-mers and edges correspond to length $k$-1 suffix-prefix overlaps between $k$-mers such that the corresponding merged ($k$+1)-mers exist in $M^{k+1}$. The later requirement is needed to ensure that both the overlapping $k$-mers are from the same DNA read. The "directed" nature of the graph mirrors the 5′-to-3′ orientation of DNA, thus graph traversal also follows the 5′-to-3′ orientation. An example is shown in Figure 4.1.

Similarly, a *bi-directed* de Bruijn graph $G = (V, E)$ (Figure 4.2), first proposed by Medvedev *et. al.* [92], is a directed graph with $k$-molecules as vertices, each vertex rep-

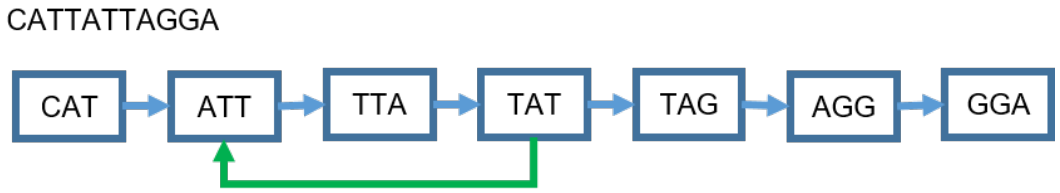



Figure 4.1: An example simple de Bruijn graph. The direction of the arrows indicate the direction of traversal.

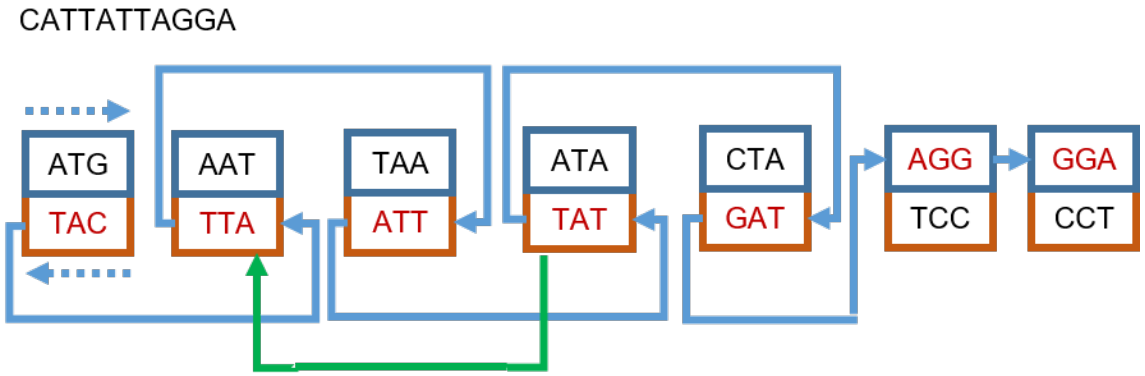



Figure 4.2: An example bi-directed de Bruijn graph for the same sequence as in Figure 4.1. The top $k$-mer in each vertex is canonical, while red denotes the original $k$-mer from the input sequence. Non-canonical $k$-mers are written in reverse lexicographical order following the $5'$ to $3'$ nature of the strand. The direction of the arrows indicate the direction of traversal.

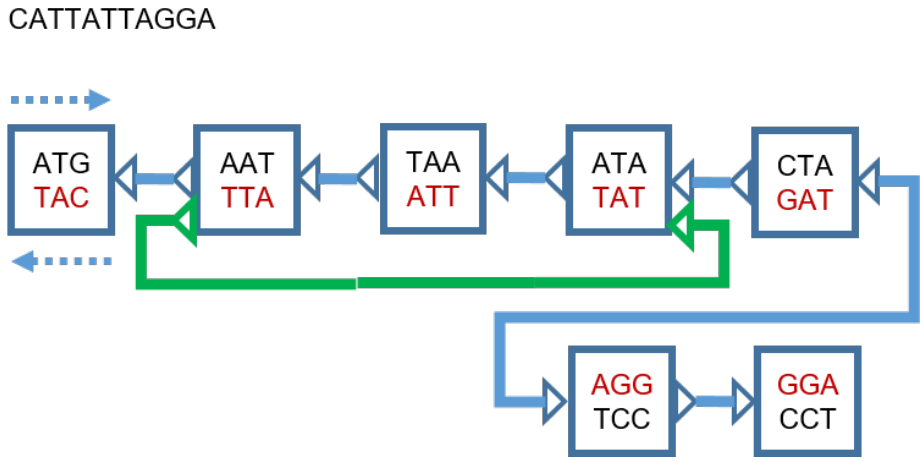



Figure 4.3: The same de bruijn graph represented using the bi-directed edge notation. Each edge is annotated with two triangles, indicating the strand used during traversal. The left side of the box represents $5'$ end of the canonical $k$-mer while the right side represents the $3'$ end, and *vice versa* for non-canonical $k$-mers. The edges connect the $3'$ end of the source $k$-mer to the $5'$ end of the target $k$-mer.

resented by its canonical $k$-mer, thus $V = \hat{M}^k$. As each vertex contains two strands, $v_x = \langle \hat{m}_x, \check{m}_x \rangle$, an edge between vertices $v_x$ and $v_y$, denoted by $(v_x, v_y) = e_{xy} \in E$, corresponds to the ($k$-1) suffix-prefix overlap between a *strand* of $v_x$ and a strand of $v_y$ in the $5'$-to-$3'$ orientation. Bi-directed de Bruijn graphs more closely model real DNA molecules and more accurately account for $k$-molecule frequencies.

In the $k$-mer notation, the strand-specific length ($k$-1) suffix-prefix overlap between $v_x$ and $v_y$ can occur in one of $4$ ways: $(\hat{m}_x, \hat{m}_y)$, $(\hat{m}_x, \check{m}_y)$, $(\check{m}_x, \hat{m}_y)$, or $(\check{m}_x, \check{m}_y)$. These corresponding commonly used bi-directed de Bruijn graph edge notation [66, 61, 62] and is illustrated in Figure 4.3.

We note that $(\check{m}_x, \hat{m}_y)$ and $(\check{m}_x, \check{m}_y)$ are equivalent to $(\check{m}_y, \hat{m}_x)$ and $(\hat{m}_y, \hat{m}_x)$, respectively, via reverse complement of the merged ($k$+1)-mers. We further note that an edge can equally be represented by a ($k$+1)-molecule, and that edges continue to embody the $5'$-to-$3'$ traversal orientation.

Bi-directed edges and the additional and necessary condition that a traversal enters and exits a vertex via the same strand together ensure that overall traversal of successive vertices and edges is via ($k$-1) suffix-prefix overlaps between $k$-mer strands in $5'$-to-$3'$ order. The graph formulation and traversal based on these conditions prevent inadvertent backtracking and circular traversals due inconsistent use of strands [66]. The resulting assembly represents the nucleotide sequence on a single DNA strand.

Previous works that compact chains in bi-directed de Bruijn graphs employ variants of the list ranking algorithm, and adopt different strategies to ensure strand consistency. Sparse ruling set based assemblers extend seed $k$-mers or $k$-molecules into chains (e.g. [65, 43]). They often incur irregular memory access and require fine grained conditional evaluation as they must track the current and matching strands during traversal. Pointer jumping based algorithms either treat the graph as directed [61] or transform the graph, for example by splitting vertices and edges by strand [66]. In these cases the advantages of bi-directed de Bruijn graphs are not realized, while the graph transformation can increase

space and computation requirements. SWAP assembler [62] employs pointer jumping using bi-directed edges and explicitly tracks the involved strands, and therefore suffers from the same fine-grained conditional evaluation overhead. In Section 4.2 we present our graph data structures and algorithm for direct compaction of bi-directed graph chains without these shortcomings.

## 4.2 Parallel Algorithm

Our parallel chain compaction algorithm supports compacting multiple bi-directed chains concurrently in the presence of cycles in distributed memory environments. Chain compaction assumes that the graph has been constructed, and proceeds in three phases: *chain vertex identification*, *vertex ordering*, and *unitig generation*. The first two phases are described in this section while the last phase is described in Section 4.3.7.

### 4.2.1 Chain Vertex Identification

The *in*- and *out*-edges of a vertex $v \in V$ are defined relative to its canonical $k$-mer representation $\hat{m} \in \hat{M}^k$. Edge $(v_x, v_y)$ is an in-edge for $v_y$ if $\hat{m}_y$ is in the second position of the corresponding $k$-mer pair, i.e., it is either $(\hat{m}_x, \hat{m}_y)$ or $(\check{m}_x, \hat{m}_y)$. We denote the in-edge of $v_y$ as $({}_y\ddot{m}_x, \hat{m}_y)$, where ${}_y\ddot{m}_x$ references the $k$-mer in $v_x$ that resides on the same strand as the chosen $k$-mer in $v_y$, $\hat{m}_y$. The out-edge of $v_y$ is similarly defined but with $\hat{m}_y$ in the first position of the pair. The in- and out-edges of $v_x$ are similarly defined.

For chain compaction, we define a *chain vertex* to have exactly one in-edge and one out-edge, and a *branch vertex* to have multiple in-edges or out-edges. A *terminal vertex* is then defined as either (type I) a vertex with exactly one in- or out-edge, or (type II) a chain vertex with at least one edge to a branch vertex.

A chain $C$ in $G$ is defined as a sequence of vertices $c_j \in V$, $0 \leq j \leq |C|$, where $|C|$ is the length of the chain. We define $c_0$ and $c_{|C|}$ to be terminal, the rest as chain vertices, and $(c_{j-1}, c_j) \in E$ for $0 < j \leq |C|$. The set of vertices from all chains is denoted by $V_c$, and

the set of pairs $(c_{j-1}, c_j)$ for all chains is denoted by $E_c$. $V_c$ can be found as the union of chain and terminal vertices in $V$, and $E_c$ consists of their in- and out-edges. The set of all chains is denoted by $\mathbb{C}$. The goal of vertex ordering is to partition $V_c$ by chain membership and to compute each vertex's position $j$ in $C$ for all $C \in \mathbb{C}$.

### 4.2.2 Concurrent Compaction

We show that all chain vertices can be processed simultaneously to compact chains independently of each other. A necessary and sufficient condition is that chains have disjoint vertex and edge sets.

**Lemma 1.** *Chains in G are disjoint.*

*Proof.* We prove by contradiction. We first show that two chains $C$ and $C'$ have disjoint vertex sets. Suppose vertex $v$ is in both $C$ and $C'$ as $c_i$ and $c'_j$ respectively. This implies one of three possibilities. Case 1: $c_i$ and $c'_j$ have distinct in-edges (or out-edges) in $C$ and $C'$, then $v$ is a branch vertex and cannot be in $C$ or $C'$. Case 2: Either or both $c_i$ and $c'_j$ are terminal vertices. Without loss of generality, let $c_i$ be terminal. Then $c'_j$ has an edge not in $C$ that can extend $C$, therefore $c_i$ cannot be terminal. Case 3: $c_i$ and $c'_j$ have identical in- and out-edges. In this case either there exists a different vertex in $C$ and $C'$ that satisfies Case 1 or 2, or all vertices in $C$ and $C'$ satisfy Case 3 and therefore $C$ and $C'$ are identical. Two chains therefore cannot share a vertex.

We next show that $C$ and $C'$ have disjoint edge sets. Since $C$ and $C'$ have disjoint vertex sets, edges between vertices in $C$ cannot also be in $C'$, and vice versa. We now show that no edge exists between vertices from the vertex sets of $C$ and $C'$. Without loss of generality, let $v$ be in $C$ and $v'$ be in $C'$ and $(v, v')$ be an out-edge of $v$. If $v$ has an out-edge in $C$, then $(v, v')$ is a second out-edge, thus $v$ is a branch vertex and cannot be in $C$. If $v$ is a type I terminal vertex without an out-edge in $C$, then the existence of $(v, v')$ contradicts $v$ as a terminal vertex. If $v$ is a type II terminal vertex without an out-edge in $C$, then $(v, v')$ implies that $v'$ is a branch vertex, thus cannot be in $C'$. Identical argument

applies for in-edge of $v$. Therefore no edge in $E$ connects $C$ and $C'$, and their edge sets are disjoint. Consequently, chains in $G$ are disjoint. □

### 4.2.3 Path Modeling

The challenges faced by previous works when compacting chains in bi-directed de Bruijn graphs stem fundamentally from the use of canonical $k$-mers to represent *both* vertices in an edge, which ultimately translate to significant heuristics [66, 62] or potentially inaccurate traversal [61].

We address the objective of maintaining strand consistency directly by using **same-strand** $k$-mer as defined in Section 4.2.1. For subsequent discussions we focus on a single chain, and refer to vertex positions in a chain by $i$, $j$, and $l$.

We define a canonical $s$-path (for *singly linked* path) using $\hat{m}_j$ to denote a path from $c_j$ to $c_l$ as:

$$\hat{p}_{jl}^{+/-} = \langle \hat{m}_j, d_{jl}, {}_j\ddot{m}_l, +/- \rangle \tag{4.1}$$

where ${}_j\ddot{m}_l$ is on the same strand as $\hat{m}_j$, $d_{jl}$ is the distance between $c_j$ and $c_l$, and the last field is $(+)$ if ${}_j\ddot{m}_l$ is $3'$ of $\hat{m}_j$ and $(-)$ otherwise. It follows that the out-edge of $c_j$ is part of the path $\hat{p}_{jl}^{+}$, and the in-edge is part of $\hat{p}_{jl}^{-}$.

We define a canonical $d$-path (for *doubly linked* path) to denote a path from $c_i$ to $c_l$ passing through $c_j$ in its canonical form $\hat{m}_j$:

$$\hat{p}_{ijl} = \langle {}_j\ddot{m}_i, d_{ij}, \hat{m}_j, d_{jl}, {}_j\ddot{m}_l \rangle \tag{4.2}$$

where $d_{ij}$ and $d_{jl}$ are distances from $c_j$ to vertices $c_i$ and $c_l$. The first $k$-mer in $\hat{p}_{jl}^{+/-}$ and the second $k$-mer in $\hat{p}_{ijl}$, and the vertices they represent, are referred to as *central* while the remaining $k$-mers (or vertices) are considered *distal*. We emphasize that the distal $k$-mers are on the same strand as a central $k$-mer. The $d$-path $\hat{p}_{ijl}$ can be viewed as the composition of $\hat{p}_{ji}^{-}$ and $\hat{p}_{jl}^{+}$ using complementary edge types (in/out) from $c_j$. Non-canonical $s$-path $\check{p}_{jl}$

and $d$-path $\check{p}_{ijl}$ which arise during our algorithm execution are defined similarly using the non-canonical $k$-mer $\check{m}_j$ of $c_j$.

By indexing $s$- and $d$-paths via their canonical central $k$-mers, the distal *same-strand* $k$-mers encode the relative orientations between $d$-paths and $s$-paths. A path containing a distal $k$-mer $\check{m}_j$ is on the opposite DNA strand as an $s$- or $d$-path with canonical central $k$-mer $\hat{m}_j$. To simplify strand switching during vertex ordering, we define the reverse complement operations for $s$- and $d$-paths.

For a $d$-path, the reverse complement operation reverses of the order of the $k$-mers and the distance fields, and then reverse complements each $k$-mer, i.e. $\overline{p}_{ijl} = \langle {}_j\overline{\ddot{m}}_l, d_{jl}, \overline{m}_j, d_{ij}, {}_j\overline{\ddot{m}}_i \rangle$. For an $s$-path, the central and distal $k$-mer remain in their positions. The reverse complement operation inverts the binary flag and reverse complements the component $k$-mers, e.g. $\overline{p^+_{jl}} = \langle \overline{m}_j, d_{jl}, {}_j\overline{\ddot{m}}_l, - \rangle$.

### 4.2.4 Vertex Ordering

During *vertex ordering*, each chain vertex is labeled with chain terminal identifiers and the distances to the terminals. Our algorithm targets distributed memory environment that consists of $P$ processors, each with private memory. The total number of vertices is denoted by $N$, $N \gg P$, and the input data is initially distributed to the processors evenly, $N/P$. We use collective communication as coarse grain synchronization between processors, and adopt the pointer doubling approach to compact chains in logarithmic number of communication rounds.

We define `canonicalize()`, which conditionally applies the reverse complement operation to $s$- and $d$-paths based on whether their central $k$-mers are canonical. We also define `map()` that assigns an $s$- or a $d$-path to a processor by the canonicalized central $k$-mer, and `send()` that moves the path tuple to the assigned processor.

Algorithm 4.1 outlines the overall algorithm, while Algorithms 4.2, 4.3, and 4.4 detail the steps involved. We refer to vertices by $k$-mers for specificity and consistency. A $d$-path

**Algorithm 4.1** Parallel Vertex Ordering

1: $d$-paths $\leftarrow$ `make_paths()` $\forall v \in V_c, e \in E_c$
2: `map`($d$-paths)
3: `send`($d$-paths)
4: $t \leftarrow 0$
5: **while** has active $d$-paths **do**
6: $s$-paths $\leftarrow$ `make_updates()` $\forall$ local $d$-paths
7: `canonicalize`($s$-paths)
8: `map`($s$-paths)
9: `send`($s$-paths)
10: `update()` local $d$-paths $\forall$ received $s$-paths
11: remove *finished* local $d$-paths
12: $t \leftarrow t + 1$
13: **end while**

referencing one terminal vertex has the corresponding distance field annotated with $*$, and is referred to as *semi-finished*. A $d$-path that references two terminal vertices is considered *finished*, and one that does not reference any terminal vertices is considered *unfinished*. A $d$-path is *active* if it is either *semi-finished* or *unfinished*.

We begin by generating $d$-paths from the chain vertices and their associated edges (Algorithm 4.1 line 1). The in- and out-edges $(c_i, c_j)$ and $(c_j, c_l)$ of vertex $c_j$ are combined to form a $d$-path in the form of Expression 4.2. Wherever an in- or out-edge is missing, the corresponding distal $k$-mer is set to be the same as the central $k$-mer and distance is set to $0$ and marked as *finished* (Algorithm 4.2). The $d$-paths are distributed to processors based on their canonical central $k$-mers via `map`() and `send`() collectively and synchronously.

**Algorithm 4.2** `make_paths()`

1: Given $c_j = \langle \hat{m}_j, \check{m}_j \rangle \in V_c$ and $({}_j\ddot{m}_i, \hat{m}_j), (\hat{m}_j, {}_j\ddot{m}_l) \in E_c$
2: **if** $({}_j\ddot{m}_i, \hat{m}_j) = \emptyset$ **then**
3: $\hat{p}_{ijl} \leftarrow \langle \hat{m}_j, 0^*, \hat{m}_j, 1, {}_j\ddot{m}_l \rangle$.
4: **else if** $(\hat{m}_j, {}_j\ddot{m}_l) = \emptyset$ **then**
5: $\hat{p}_{ijl} \leftarrow \langle {}_j\ddot{m}_i, 1, \hat{m}_j, 0^*, \hat{m}_j \rangle$.
6: **else**
7: $\hat{p}_{ijl} \leftarrow \langle {}_j\ddot{m}_i, 1, \hat{m}_j, 1, {}_j\ddot{m}_l \rangle$.
8: **end if**
9: return $\hat{p}_{ijl}$

We then iteratively *compact* until all chain $d$-paths are marked as *finished*. First, two

$s$-paths are generated from the distal $k$-mers and the distances of each $d$-path (Algorithm 4.3). An $s$-path is marked as finished if either the source in- or out-distance is marked as finished. The $s$-paths are canonicalized and sent to processors based on is "central" $k$-mers, using the same processor assignment function as that for the $d$-paths in order to collocate $s$-paths and $d$-paths by the same "central" $k$-mers.

**Algorithm 4.3** `make_updates()`

Given $d$-path $\langle {}_j\ddot{m}_i, d_{ij}, \hat{m}_j, d_{jl}, {}_j\ddot{m}_l \rangle$
$d_{il} \leftarrow d_{ij} + d_{jl}$
$d_{li} \leftarrow d_{ij} + d_{jl}$
**if** $d_{ij}$ is marked as finished **then**
    mark $d_{li}$ as finished
**end if**
**if** $d_{jl}$ is marked as finished **then**
    mark $d_{il}$ as finished
**end if**
append $\langle {}_j\ddot{m}_i, d_{il}, {}_j\ddot{m}_l, + \rangle$ to $s$-paths
append $\langle {}_j\ddot{m}_l, d_{li}, {}_j\ddot{m}_i, - \rangle$ to $s$-paths

Each processor receives up to two $s$-paths for each of its $d$-paths, one for the $5'$ distal $k$-mer, and the other for the $3'$ distal $k$-mer. The $d$-path is updated by replacing the appropriate distal $k$-mer and distance. (Algorithm 4.4). Once the updates are completed, $d$-paths marked as *finished* are excluded from further processing. The process continues until all $d$-paths corresponding to chain vertices are marked as *finished*, the mechanism through which this is measured is described in Section 4.2.6.

**Algorithm 4.4** `update()`

Given $d$-path $\langle {}_j\ddot{m}_i, d_{ij}, \hat{m}_j, d_{jl}, {}_j\ddot{m}_l \rangle$
Given $s$-path $\langle \hat{m}_x, d_{xy}, {}_x\ddot{m}_y, flag \rangle$, $\hat{m}_x \equiv \hat{m}_j$
**if** $flag == +$ **then**
    update $d$-path to $\langle {}_j\ddot{m}_i, d_{ij}, \hat{m}_j, d_{xy}, {}_j\ddot{m}_y \rangle$
**else if** $flag == -$ **then**
    update $d$-path to $\langle {}_j\ddot{m}_y, d_{xy}, \hat{m}_j, d_{jl}, {}_j\ddot{m}_l \rangle$
**end if**

We make the following observations about our parallel chain compaction algorithm. First, the algorithm labels and ranks along both DNA strands simultaneously. At comple-

tion, each $d$-path has distal $k$-mers corresponding to both terminal vertices of a chain. Second, $d$-paths for neighboring vertices may reference opposite strands of a chain molecule. Prior to unitig generation, the $d$-paths are converted to use the strand on which the chain representative $k$-mer resides, which is defined as the lexicographically smaller of the two $5'$ terminal $k$-mers. We note that the algorithm applies to directed de Bruijn graph if initially the $k$-mers and ($k$+1)-mers are not canonicalized, and canonicalization of the $d$- and $s$-paths are bypassed.

### 4.2.5 Correctness

We now show that our vertex ordering algorithm produces the correct results and in finite time. By Lemma 1, We consider vertex ordering for a single chain.

**Theorem 1.** *The parallel* vertex ordering *algorithm produces the* ordered and labeled $d$-*path* $\langle {}_j\ddot{m}_0, d_{0j}, \hat{m}_j, d_{j|C|}, {}_j\ddot{m}_{|C|}\rangle$ *for vertex* $c_j$, $0 \leq j \leq |C|$, *from chain* $C$.

*Proof.* The proof is by induction. Consider $\{c_j\}$ as the set of vertices on chain $C$, and $c_0$ and $c_{|C|}$ as the chain's terminal vertices. Let $\hat{m}_j$ be the canonical $k$-mer associated with vertex $c_j$ and let $0 \leq i \leq x \leq j \leq y \leq l \leq |C|$.

Base Case: At $t = 0$, the terminal vertex $c_0$ is represented by $d$-path $\hat{p}_{001} = \langle m_0, 0^*, m_0, 1, {}_0\ddot{m}_1\rangle$ if $m_0$ is canonical and $\hat{p}_{100} = \langle {}_0\ddot{m}_1, 1, \overline{m}_0, 0^*, \overline{m}_0\rangle$ otherwise, and similarly for $c_{|C|}$. These $d$-paths are marked as *semi-finished*. All other paths $\{\hat{p}_{(j-1)j(j+1)}\}$, $0 < j < d$, are *unfinished*.

Induction Hypothesis: At the end of iteration $t-1$, $t \geq 1$, vertex $c_j$ is associated with $\hat{p}_{xjy}$. The distance $d_{xj} < 2^t$ and $\hat{m}_x \equiv \hat{m}_0$ if $d_{xj}$ is marked as *finished*, and $d_{xj} \equiv 2^t$ and $\hat{m}_x \neq \hat{m}_0$ otherwise. Similarly, $d_{jy} < 2^t$ and $\hat{m}_y \equiv \hat{m}_d$ if $d_{jy}$ is marked as *finished*, and $d_{jy} \equiv 2^t$ and $\hat{m}_y \neq \hat{m}_d$ otherwise.

Induction Step: During iteration $t$, two $s$-paths $\langle {}_x\ddot{m}_i, d_{ij}, {}_x\ddot{m}_j, +\rangle$ and $\langle {}_x\ddot{m}_j, d_{ij}, {}_x\ddot{m}_i, -\rangle$ are generated from $\hat{p}_{ixj}$ with $d_{ij} = d_{ix} + d_{xj}$ and same-strand $k$-mers relative to $\hat{m}_j$. After canonicalization and communication, the $s$-path becomes $\langle \hat{m}_i, d_{ij}, {}_i\ddot{m}_j, flag\rangle$, and is used

to update the $d$-path for $c_i$ to $\langle {}_i\ddot{m}_\bullet, d_{\bullet i}, \hat{m}_i, d_{ij}, {}_i\ddot{m}_j \rangle$ if $flag \equiv +$, and $\langle {}_i\ddot{m}_j, d_{ij}, \hat{m}_i, d_{i\bullet}, {}_i\ddot{m}_\bullet \rangle$ otherwise. Similarly $\langle {}_x\ddot{m}_j, d_{ij}, {}_x\ddot{m}_i, - \rangle$ updates $d$-path for $c_j$.

Case 1: The distances $d_{ix}$ and $d_{xj}$ are not marked as finished. By the induction hypothesis, $d_{ix} \equiv d_{xj} \equiv 2^t$, and $\hat{m}_i$ and $\hat{m}_j$ do not represent terminal vertices. The distance $d_{ij}$ then is $2^t + 2^t = 2^{t+1}$ and is not marked as ***finished***, and the updated $d$-paths for $c_i$ and $c_j$ have distal $k$-mers that do not represent terminal vertices.

Case 2: One of $d_{ix}$ and $d_{xj}$ is marked as ***finished***. Without loss of generality, let $d_{ix}$ be marked ***finished***. By the induction hypothesis, $d_{ix} < 2^t$ and $d_{xj} = 2^t$, $\hat{m}_i$ is terminal and $\hat{m}_j$ is not. The new distance is then $d_{ij} < 2^{t+1}$. The $d$-path for $c_i$ is updated with a distal $k$-mer $m_j$, which is not terminal and therefore $d_{ij}$ is not marked as ***finished***. The $d$-path for $c_j$, however, is updated with a terminal distal $k$-mer $m_i$, thus $d_{ij}$ is marked in this case. The same logic applies when $d_{xj} < 2^t$ instead.

Case 3: Both $d_{ij}$ and $d_{jl}$ are marked as finished. Then the algorithm does not update this $d$-path further as it is ***finished*** with $d$-path $\langle {}_j\ddot{m}_0, d_{0j}, \hat{m}_j, d_{j|C|}, {}_j\ddot{m}_{|C|} \rangle$ by the induction hypothesis.

Identical process generates $s$-paths from the $d$-paths $p_{jyl}$, which are then used to update $p_{\bullet jy}$ and $p_{yl\bullet}$ in the same manner. □

**Corollary 1.** *The parallel* vertex ordering *algorithm completes in* $\lceil \log(|C|_{max}) \rceil$ *iterations, where* $|C|_{max}$ *is the length of the longest chain in* $G$.

*Proof.* As shown in the proof for Theorem 1, during each iteration, at least one of the distances in $d$-path $\langle {}_j\ddot{m}_i, d_{ij}, \hat{m}_j, d_{jl}, {}_j\ddot{m}_l \rangle$ of vertex $c_j$ doubles in length to $2^{t+1}$, except during the last iteration when all $d$-paths are marked ***finished***. The longest chain with $|C|_{max}$ edges has as its $3'$ terminal vertex $c_{|C|_{max}}$ with $\hat{p}_{i|C|_{max}|C|_{max}}$, whose $5'$ distance field is marked finished at iteration $t = \lceil \log(|C|_{max}) \rceil$. Since our algorithm processes all chains and their vertices concurrently, it completes in $\lceil \log(|C|_{max}) \rceil$ iterations.

□

### 4.2.6 Cycle Detection

A special case occurs when a chain in the graph forms a cycle. As there are no terminal vertices in a cycle, Algorithm 4.3 cannot mark any distances as finished and Algorithm 4.1 continues indefinitely. We therefore seek a mechanism to identify vertices on a cycle during *vertex ordering* iterations.

Cycles arise naturally in genomic data, and are handled differently by each chain compaction and assembly tool. As a library, Bruno supports the identification and extraction of cycle vertices for downstream application use. As stated in Section 1.1.2, HipMer, Zeng's assembly algorithm, and BCALM2 are unable to identify cycle vertices. ParBiConstruct [66] defines its list ranking stopping criteria as the non-decreasing count of merged tuples, which implicitly identifies cycle vertices.

This termination criteria is not suitable for the bidirectional traversal employed by our compaction algorithm. Each $d$-path continues to merge with others until it encompasses the entire chain. The *join count* for one chain remains constant before iteration $t = \lceil \log(|C|_{max}) \rceil$, when the entire chain is compacted. To prevent premature termination, at least one chain must be compacted completely during each iteration.

Instead, we use the number of *unfinished* $d$-paths in designing the stopping criterion in the presence of cycles. We first formally describe the behavior of the $d$-paths for cycle and chain vertices. We then describe their application to the design of the stopping criterion and the formal identification of cycle vertices.

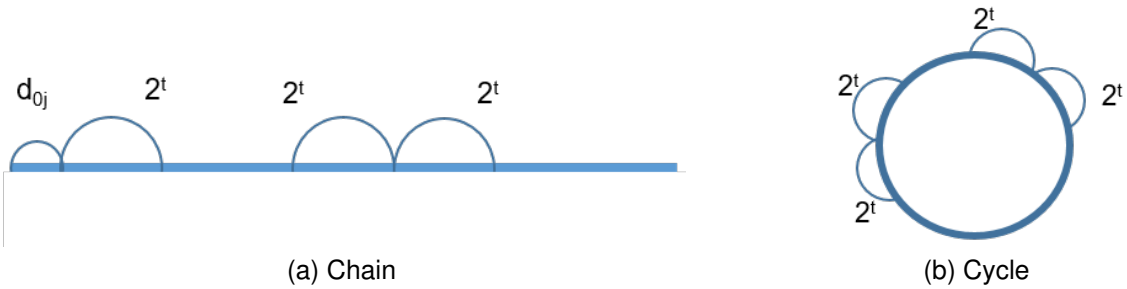



Figure 4.4: Illustration of $d$-path distances for chain (4.4a) and cycle (4.4b) vertices.

**Corollary 2.** *All cycle $d$-paths remain* unfinished *for all iterations of* vertex ordering *process, and contain distances $2^t$ at the start of iteration $t$.*

*Proof.* The proof can be derived as a special case from the proof for Theorem 1. In the absence of terminal vertices, all $d$-paths or cycle vertices are initialized as *unfinished* for the Base Case. The Induction Hypothesis implies that all cycle $d$-paths are *unfinished* and have distances $2^t$ during iteration $t$. During the Induction Step, only Case 1 is applicable and therefore all $s$-paths are marked as *unfinished* and the distances in the $d$-paths are updated to $2^{t+1}$ for the next iteration. □

**Lemma 2.** *The number of* unfinished *$d$-paths on a chain strictly decreases to zero as* vertex ordering *iterations progress.*

*Proof.* Let $\{c_j\}$ be the set of vertices on chain $C$, $0 \leq j \leq |C|$. By Corollary 1, $\lceil \log(|C|) \rceil$ iterations are required for vertex ordering. We initialize the corresponding $d$-paths according to Algorithm 4.2. We now examine the *unfinished* $d$-path count for iteration $t$.

Case 1: For iterations $0 < t < \lceil \log(|C|) \rceil - 1$, $2^t \leq \frac{|C|}{4}$. The $d$-paths for $\{c_j \mid 0 \leq j < 2^t\}$ are initially marked as *semi-finished*. The $d$-paths for $\{c_l \mid l = j + 2^t, 0 \leq j < 2^t\}$ are updated by $s$-path from $\{c_j\}$ to become *semi-finished*. Since $2^t \leq \frac{|C|}{4}$ and $l < \frac{|C|}{2}$, the $s$-path updates do not cross the midpoint of the chain. The same logic applies for $\{c_j \mid (|C| - 2^t) < j \leq |C|\}$ and $\{c_l \mid (|C| - 2^{t+1}) < l \leq (|C| - 2^t)\}$ by symmetry. During iteration $t$, $d$-paths in the ranges $[2^t, 2^{t+1})$ and $(|C| - 2^{t+1}, |C| - 2^t]$ are updated from *unfinished* to *semi-finished*, while the $d$-paths in range $[2^{t+1}, |C| - 2^{t+1}]$ remain as *unfinished*. Therefore the *unfinished* $d$-path count decreases by $2^{t+1}$ during iteration $t$.

Case 2: For iteration $t = \lceil \log(|C|) \rceil - 1$, $\frac{|C|}{4} < 2^t \leq \frac{|C|}{2}$. The $d$-paths for $\{c_l \mid 2^t \leq l \leq \frac{|C|}{2}\}$ are previously unmarked and are updated by $s$-paths from vertices $\{c_j \mid 0 \leq j \leq \frac{|C|}{2} - 2^t\}$. Similar argument applies to $\{c_l \mid \frac{|C|}{2} \leq l \leq |C| - 2^t\}$. Therefore, all *unfinished* $d$-paths in the range $l \in [2^t, |C| - 2^t]$ are updated to *finished* or *semi-finished*. The *unfinished* $d$-path count reduces to zero during this iteration.

Case 3: Iteration $t = \lceil \log(|C|) \rceil$. All $d$-paths are previously marked as either *semi-finished* or *finished*. The number of *unfinished* chain $d$-paths remains zero.

Since the number of *unfinished* $d$-paths decreases by $2^{t+1}$ during each iteration in case 1, reduces to zero during case 2, and remains zero during case 3, it strictly decreases to zero. □

Corollary 2 and Lemma 2 are illustrated in figure 4.4. We can infer that the total number of *unfinished* $d$-paths strictly decreases until only *unfinished* $d$-paths for cycle vertices remain. Our algorithm leverages this behavior as the stopping criterion. We compute the number of *unfinished* $d$-paths by counting $d$-paths with $2^t$ in both distance fields. The total number is obtained via a global reduction. The algorithm terminates when the total *unfinished* count remains constant. We note that this process identifies cycle vertices but does not distinguish between cycles. Such requirement can be satisfied via an additional connected components labeling step after *vertex ordering*. Alternatively, the lexicographically minimal $k$-mer seen during *vertex ordering* can be used to annotate vertices as a cycle identifier, at the cost of a constant factor increase in communication volume, storage, and computation time, without changing the algorithm complexity.

### 4.3 Implementation

We implemented our algorithm using C++ and MPI, and leveraged the mxx library [78] as the C++ API wrapper for MPI and for parallel sample sort implementation. We used the Kmerind $k$-mer indexing library [39] for its alphabet, $k$-mer, file I/O, count index, and distributed hash tables implementations. The $k$-mers and ($k$+1)-mers are encoded using 2 bits per character. We store only canonical $k$-mers and ($k$+1)-mers, as they compactly represent $k$- and ($k$+1)-molecules. Data located within a processor is referred to as "local". Distributed hash tables are used for storage, query, and communication of $k$-mers and associated data, such as $k$-mer neighbors and distances. We use Google Dense Hashmap for local storage and FarmHash for local hash table and to map $k$-mers to processors in

`map()`. Two new operations were added to Kmerind indices, `update` and `exists`, to facilitate Bruno implementation. They have algorithms and complexity similar to `insert` and `count`, respectively.

Our implementation consists of $5$ basic steps: *file parsing*, *graph construction*, *chain vertex extraction*, *vertex ordering*, and *unitig generation*. The first two steps form the *construction* phase, while the last three steps are referred to as the *compaction* phase. Error detection and removal in the de Bruijn graph is described in Chapter 5. Section 4.2.4 focuses on the chain compaction step.

### 4.3.1 File Parsing

We begin by reading the read set files in parallel on $P$ processors using the FASTQ and FASTA file readers provided by Kmerind. Each processor receives approximately $F/P$ bytes, where $F$ is the total file size. The file data is stored in a byte array in memory as our implementation parses this data twice. During subsequent parsing, reads containing "N" are ignored.

### 4.3.2 Graph Vertex and Edge Representation

A canonical bi-directed de Bruijn graph $G$ consists of a distributed hash table with tuples $\langle \hat{m}, f \rangle$, where $\hat{m}$ is the canonical $k$-mer representing a vertex and serving as key in the map, and $f$ is the set of frequencies corresponding to each edge and the $k$-mer itself:

$$f = \langle f_{\hat{m}A}, f_{\hat{m}C}, f_{\hat{m}G}, f_{\hat{m}T}, f_{A\hat{m}}, f_{C\hat{m}}, f_{G\hat{m}}, f_{T\hat{m}}, f_{\hat{m}} \rangle$$

The frequency values correspond implicitly to the 5' and 3' edges by position,in the order of 5' A, C, G, T, followed by 3' A, C, G and T edges. The final value $f_{\hat{m}}$ represents the $k$-mer frequency. For applications that only require knowledge of the presence or absence of an edge, $f$ can be compacted into 8-bits and the $k$-mer presence in the hash table indicates

overall presence. This representation is identical to that used by ABySS [8] and similar to that in HipMer [63]. For applications that require knowledge of the edge frequency values, $f$ is a $9$-tuple with elements of the unsigned integer type in user specified bit width.

### 4.3.3 Graph Construction

We define a compact representation for the edges of a vertex $k$-mer, denoted by $e$ and encoded in a single byte. The upper and lower $4$ bits represent the incoming and outgoing edge sets respectively, and each bit in a $4$ bit group correspond to a letter in the DNA alphabet, $\{$A, C, G, T$\}$. A bit is set if the concatenation of the $k$-mer and the alphabet letter forms a ($k$+1)-mer in the read set.

To construct the graph, we locally parse the file data into tuples of the form $\langle k$-mer, $e\rangle$, where at most $1$ bit is set in each of the upper and lower $4$-bit groups. Kmerind's parsers are used, to generate the tuples in linear time in a sliding window manner.

Graph construction then involves insertion of the array of $\langle k\text{-mer}, e\rangle$ into a distributed hash table and updating the corresponding frequency fields where $e$ has a bit set. We note that if $f$ tracks presence and absence, $e$ and $f$ can be combined via bitwise OR.

If the $k$-mer is canonical, then the tuple is inserted directly, otherwise the reverse complement of the $k$-mer and $e$ are inserted. The reverse complement of $e$ is equivalent to the reversal of the bits in $e$. As this process requires communication, the time complexity of the graph construction under uniform $k$-mer distribution assumption is $O(N/P + \tau p + \mu N/p)$ [39], where $N$ is the total number of ($k$+1)-mers and $\tau$ and $\mu$ are latency and throughput coefficients for communication. We promote uniform distribution via appropriate choice of hash functions, such as MurmurHash and Google FarmHash.

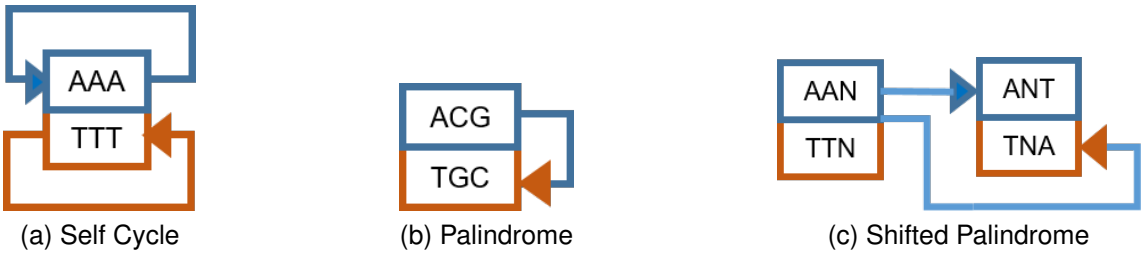
### 4.3.4 Chain Vertex Extraction

After the graph is constructed, the chain vertices are extracted. We first identify vertices with in- or out-degrees of at most $1$ via a linear scan of the local graph hash table.

The selected tuples $\langle \hat{m}_j, e \rangle$ are converted to $d$-paths $\langle {}_j\ddot{m}_i, d_{ij}, \hat{m}_j, d_{jl}, {}_j\ddot{m}_l \rangle$ based on Algorithm 4.2. The upper (and lower) $4$ bits of $e$ are converted to the corresponding alphabet character and prepended (and appended) to the ($k$-1)-prefix (and -suffix) of $\hat{m}_j$ to generate ${}_j\ddot{m}_i$ (and ${}_j\ddot{m}_l$). If none of the $4$ bits are set, then the edge is considered absent in $E_c$. The central $k$-mer in canonical form, $\hat{m}_j$, serves as hash key and the remaining fields as value. The same hash function is used for the distributed graph and $d$-path hash tables to ensure identical $k$-mer-to-processor assignment.

In the second step, we mark $d$-paths with distal $k$-mers that represent type II terminal vertices by setting the corresponding distance to $0$. We first select the branch vertices via a local linear scan of the graph hash table. For each branch vertex $m_j$, its incoming and outgoing edges are transformed to $s$-paths $\langle {}_j\ddot{m}_i, 0^*, \hat{m}_j, + \rangle$ and $\langle {}_j\ddot{m}_l, 0^*, \hat{m}_j, - \rangle$ respectively. The $s$-paths are canonicalized and distributed as updates to the $d$-path hash table based on their first $k$-mers. If the target $k$-mer does not exist in the $d$-path hash table, then the $s$-path is ignored. Otherwise the distance $d_{ij}$ or $d_{jl}$ is set to zero depending on the orientation flag.

Certain $k$-mer parameters and input data can create vertices that satisfy the chain vertex definition, yet require special care to handle. Five such vertex types are *isolated*, *self cycle*, *palindrome*, *shifted palindrome*, and *cycle* vertices. The first $4$ are illustrated in Isolated vertices have in- and out-degrees of $0$ and are trivial to detect and exclude. Cycle vertices are identified using the approach described in Section 4.2.6. The edges of a self-cycle vertex reference the vertex itself, and can be handled by the cycle detection process.



(a) Self Cycle (b) Palindrome (c) Shifted Palindrome

Figure 4.5: Examples of *self cycles*, *palindrome* and *shifted palindrome*. In 4.5b, the palindromic sequence is ANT. Each of these local structures can result in cyclic traversal or change in traversal directions.

*Palindrome* arises when a $k$-mer is its own reverse complement. Similarly a ($k$+1)-mer being its own reverse complement, for example "ACGT", results in a *shifted palindrome* for its prefix and suffix $k$-mers, "ACG" and "CGT". The presence of these structures causes the traversal direction to reverse. We treat these palindromic structures as branches and exclude them from the $d$-path hash table, as well as marking their neighbors as terminal $d$-paths.

### 4.3.5 Vertex Ordering

The *vertex ordering* process follows closely the steps outlined in Algorithms 4.3 and 4.4, executed iteratively as described in Algorithm 4.1. The iteration begins with the construction of an array of pointers to *unfinished* and *semi-finished* $d$-paths in the $d$-path hash table via a local linear scan.

During an iteration, the $d$-path hash table entries are accessed via the pointers in the array and two $s$-paths are constructed for each $d$-path: $\langle m_i, d_{il}, {}_i\ddot{m}_l, + \rangle$ and $\langle m_l, d_{il}, {}_l\ddot{m}_i, - \rangle$. The distances in the $d$- and $s$-paths are represented as a 32-bit signed integer, as a chain is unlikely to contain more than than $2^{31}$ vertices. The sign bit is used to indicate that the distal $k$-mer references a chain terminal vertex in its third field. The $d$-path hash table entries are updated according to Algorithm 4.4 via local hash table queries.

At the end of the iteration, the local pointer array is linearly scanned and pointers to *finished* $d$-paths are deleted. We perform a global reduction of the number of remaining *unfinished* $d$-paths to determine if all chain vertices have been compacted. The *vertex ordering* process terminates when the *unfinished* count across all processors reaches $0$ or stops decreasing, as discussed in Section 4.2.6.

### 4.3.6 Optimized Vertex Ordering

An optimization to our *vertex ordering* algorithm is based on the observation that during an iteration $t > 1$, the $d$-paths within a distance of $2^t$ of an chain terminal $d$-path are

*semi-finished* and each will generate an $s$-path for updating the chain terminal $d$-path. The chain terminal $d$-path is updated using the $s$-path with the maximum distance, which is $min(2^t, |C|)$, $|C|$ being the length of chain $|C|$. The remaining $2^t - 1$ $s$-paths produce no useful result while requiring communication and computation. We therefore limit the generation of $s$-paths from *semi-finished* $d$-paths: If a $d$-path is *semi-finished*, then it can generate a chain terminal update $s$-path only if the sum of its distances is $2^{t+1}$.

This optimization reduces communication volume and computation by $2^t - 1$ for each terminal $d$-path in each chain during each iteration $t$. The reduction in communication volume per chain over the entire *vertex ordering* process is approximately $\Sigma_{t=1}^{\log(|C|)} 2 * (2^t - 1) = 4|C| - 2 - \log^2(|C|) - \log(|C|)$, and the total communication reduction is $\Sigma_{C \in \mathbb{C}}(4|C| - 2 - \log^2(|C|) - \log(|C|))$, where $\mathbb{C}$ is the set of chains. We note that the minimum for the expression $4|C| - 2 - \log^2(|C|) - \log(|C|)$ for all $|C| \geq 1$ occurs at $|C| = 1$. Therefore, while the distribution of chain lengths affects the amount of communication reduction, the optimization is beneficial for all chain length distributions.

### 4.3.7 Unitig Generation

After *vertex ordering*, each vertex $c_j$ has $d$-path $\langle {}_j\ddot{m}_i, d_{ij}, \hat{m}_j, d_{jl}, {}_j\ddot{m}_l \rangle$ containing $k$-mers of its chain's two terminal vertices as ${}_j\ddot{m}_i$ and ${}_j\ddot{m}_l$. Note that ${}_j\ddot{m}_i$ is $5'$ to $\hat{m}_j$ and on the same strand, and therefore it is a $5'$ $k$-mer for the path. Similarly, ${}_j\ddot{m}_l$ is the $3'$ $k$-mer for the path. We choose the lexicographically smaller of the two $5'$ $k$-mers, i.e. ${}_j\ddot{m}_l$ or ${}_j\overline{\ddot{m}}_l$, as the chain's representative $k$-mer. The chain representative $k$-mer can be computed locally for each $d$-path without communication.

We sort the chain $d$-paths with the chain representative $k$-mer as the primary key and the corresponding distance as the secondary key to group $d$-paths by chain id, and to order the $d$-paths within each chain. Parallel sample sort with bitonic sample sorting from the mxx library [78] is used for this task. Each terminal vertex that represents a chain has its complete $k$-mer written to a byte array, while for the remaining vertices on the chain the

last character, or the reverse complement of the first character, is written to the byte array, ordered by distance from the chain representative vertex. The byte array is then written to disk via MPI-IO.

## 4.4 De Bruijn Graph Data Structure and API

To standardize and simplify interaction with de Bruijn graph, we define a set of foundational operations for a distributed de Bruijn graph and its compacted chains. We first define the data structures, and then discuss the operations at high level. For the remainder of this document, we assume the hash table uses canonical $k$-mers as key, but it is not a requirement of the implementation.

### 4.4.1 De Bruijn Graph Data Structures

The de Bruijn Graph representation (Section 4.3.2) allows the frequencies of each edge and vertex to be tracked. Traversal through a vertex enters through an in-edge and exists out of an out-edge. Such a traversal correspond to a ($k$+2)-mer, potentially in the input sequence data. Frequency of ($k$+2)-mers can therefore be related to the edge frequencies in a graph vertex. While recording each ($k$+2)-mer frequency passing through a graph vertex can reduce some ambiguity during traversal through a branch vertex, $25$ values must be stored ($\{A, C, G, T, \emptyset\} \times \{A, C, G, T, \emptyset\}$) instead of $8$ ($\{A, C, G, T\} + \{A, C, G, T\}$). Storing only edge frequencies, corresponding to ($k$+1)-mer frequencies represent a balanced compromise.

The result of the chain compaction consists of a distributed hash table of $d$-paths (Eq. 4.2), each instance representing a chain vertex, its left and right neighbors, and corresponding distances. The set of chains is denoted $\mathbb{C}$. This hash table is referred to as a *chain map*. The central $k$-mer serves as the hash table key, while the remaining fields are the mapped values.

The chain map contains only topological information while $G$ contains metadata in-

cluding edge and $k$-mer frequency. The pair $\langle G, \{C\}\rangle$ is considered the compacted de Bruijn graph. As $G$ and $\mathbb{C}$ are distributed, we require both to use the same upper level distribution hash function to ensure that frequency and topology information for a chain vertex are co-located on the same processor in order to reduce communication.

Operations on $G$ and $\mathbb{C}$ are approximately categorized into three levels based on the required context of the operations. We note that all operations operate in batch in order to minimize the impact of communication latency, which is incurred per distributed operation invocation.

### 4.4.2 Level 1 Operations

Level 1 operations operate on individual vertices and edges in $G$ and $d$-paths in $\mathbb{C}$ individually. These include the basic Kmerind hash table operations that are random access in nature: `insert`, `find`, `erase`, `update`, and `exists`.

Graph-specific specializations for the `erase` operation, `erase_verties`, are provided for $G$. When a target vertex $v$ is deleted, its out edges $\{\langle v, u\rangle\}$ are deleted as well. To maintain consistency of the graph and correctness of traversal, the in-edge $\langle v, u\rangle$ in the neighbor vertex $u$ is also deleted. Similarly for the out-edges of $v$.

Additionally, for finding and modifying edges in $G$, we also define `find_edges` and `erase_edges` operations. For these operations, the matching vertices are located first and then the target edge frequencies in $\langle \hat{m}, f\rangle$ are returned or modified. For $\mathbb{C}$, the `update` operation provides the same functionality for modifying the edges.

All of the Level 1 hash table operations except for `insert` include 3 variants. The first accepts a vector of $k$-mers or $k$-mer edge pairs, and operate on matching vertices and edges in the hash tables. The second variant accepts a user supplied predicate function that is applied to all vertices and/or edges that satisfy the predicate. This variant does not require communication. The third variant accepts the target vertex/edge vector as well as the predicate function, and the matching vertices or edges that also satisfy the predicate are

processed.

A set of simple predicates have been developed as part of the implementation of the algorithms described in this work. These include predicated to identify chain and branch vertices in $G$ and terminal and internal chain vertices in $\mathbb{C}$.

Finally, we consider parallel sorting with user-defined comparison operators as part of the set of Level 1 operations. Sorting establishes an ordering based on pair-wise relationship between vertices in $G$ and $\mathbb{C}$ in order to facilitate higher level and application-defined operations.

**COMPLEXITY ANALYSIS**: The hash table based Level 1 operations processes the input in a single or a constant number of passes. Their complexities for computation and communication therefore directly correspond to the Kmerind Hashed Uni-index complexities outlined in Sections 2.1.3 and 2.1.3.

We utilize parallel sample sort for distributed memory sorting. Its complexity is described in Section 2.1.4

*Example: Retrieve Branch Vertices*

A branch vertex in $G$ has more than 1 in-edge, or more than 1 out-edge. This criteria can easily be implemented as a predicate function. Using the predicate function with the `find` operation returns all branch vertices in $G$. Chain vertices can be identified similarly, and is used for initial population of the chain map prior to compaction.

*Example: Graph Traversal*

Given a vertex in $G$, we can obtain the $k$-mers representing its neighbors locally by shifting the vertex's $k$-mer and prepending or appending single characters corresponding its neighbors. The neighbor $k$-mers can then be used as input parameter for the distributed `find` operation to retrieve the corresponding vertex and metadata, which further allows the vertices at distance 2 to be constructed. The graph can therefore be traversed iteratively.

*Example: Compacted Graph Traversal*

Compacted graph can be similarly traversed, with the advantage that the chain map can be used to jump to the distal end of a chain. When a chain vertex is encountered in the $G$, the chain map can be queried via the distributed `find` operation to retrieve the corresponding $d$-path. The chain terminal $k$-mer can then be extracted from $d$-path in constant time and used for further traversal in $G$.

### 4.4.3 Level 2 Operations

Level 2 operations focus primarily on individual compacted chains in $G$ and $\mathbb{C}$. As the vertices in $G$ and $\mathbb{C}$ are distributed, vertices on a chain can be scattered to multiple processors. Level 2 operations therefore involves re-constructing spatial locality, or strict ordering, and generating derived chain metadata after reconstruction. In addition to the two examples listed below, generating a compressed representation of chains (Section 5.4.1) also is a Level 2 operation.

Level 2 operations depend heavily on the Level 1 operations to query for matching chain vertices in $\mathbb{C}$ and parallel sample sort with application-defined comparison operators for vertex ordering. After the spatial locality or order has been established, a local linear scan is sufficient for generating the derived data.

**COMPLEXITY ANALYSIS**: Query for matching chain vertices has the same complexity as the distributed hash table query, and with global predicate it reduces to a local hash table query with complexity $O(N/p)$. Complexity of the parallel sample sort has $O(M \log(M))$ for the local sort and $O(p \log^2(p))$ for bitonic sort of the splitters, and $O(\tau \log(p) + \mu|M| \log(p))$ for communication. Post-sorting processing has complexity dependent on the operation at hand. Often times these are simple summary operations that require only linear scans.

*Example: Unitig Generation*

The unitig generation process described in Section 4.3.7 is a direct example of a Level 2 operation. It leverages parallel sorting, a Level 1 operation, to establish grouping (by chain) and ordering (by distance to chain representative), and the emit strings corresponding to the chains via a linear scan of the ordered vertex list.

*Example: Summarizing Frequencies of Chains*

To select or filter chains based on frequency, it is useful to summarize the frequencies of the $k$-mers on the chains, for example with Gaussian distribution assumption. In this case, the mean, standard deviation, minimum, and maximum can be computed.

The algorithm begins with extraction of chain vertices using the global predicate variant of the Level 1 `find` operation on $G$. The $k$-mers and their frequencies are are then extracted and inserted into a local hash table with $k$-mer as key, each with a place holder for the chain representative. The elements in the chain map are extracted and used to update the chain representative $k$-mer fields in the local hash table. Vertices of the same chain are grouped together with a parallel sort of the contents of the local hash tables, using the chain representative $k$-mer for the comparison operator. Finally, a local linear scan is used to summarize the $k$-mer frequencies for each chain.

### 4.4.4 Level 3

Level 3 operations include graph-wide operations that are designed to support specific class of applications, such as assembly. Algorithms discussed in this chapter, including *vertex ordering*, *cycle detection*, *pre-construction error filtering*, and those discussed in Chapter 5 including *bubble* and *dead end* removal, are designated Level 3 operations. We note that Level 3 operations may be composed from Level 1 and 2 operations, or may include application-specific logic. The details of these operations are not repeated here.

## 4.5 Summary

In this chapter we presented the de Bruijn graph construction and chain compaction algorithms in the Bruno library.

We presented the chain compaction algorithm in detail, and proved its correctness, convergence, and run time complexity. We formally showed that multiple chains in a de Bruijn graph can be compacted concurrently, and introduced the same-strand $k$-mer notion for intermediate $s$- and $d$-paths, that, when coupled with path canonicalization, greatly simplify the chain compaction process while ensuring traversal direction consistency in bi-directed de Bruijn graphs. Our algorithm requires logarithmic rounds of bulk synchronous communication, while our symmetric $d$-path representation allows pointer doubling between communication rounds without additional remote queries. Finally, we presented a mechanism to detect the presence of vertices from cycles in de Bruijn graphs during the chain compaction process, that can be further extended to label cycles with a small constant factor computational, space, and communication overhead.

Finally, we present Bruno's API design, consisting of 3 levels corresponding to operations on vertices and edges, chains, and the overall graph.

# CHAPTER 5
# ERROR REMOVAL IN DE BRUIJN GRAPHS

The Bruno library currently supports parallel construction of de Bruijn graphs as well as parallel chain compaction for short read sequences. Sequencing technologies are not perfect, however. The sequencing process may introduce errors in the sequence data, in the forms of incorrectly called nucleotide bases (substitutions), missing nucleotide bases (deletions), or extra nucleotide bases (insertions). The common observation is that short read sequencers such as Illumina's systems have an error rate of approximately $1\%$, majority of which are substitutions, while long read sequencers such as PacBio and Oxford NanoPort have error rates in the $15\%$ to $20\%$ range with the majority being insertions and deletions.

As $k$-mers are constructed using a sliding windows of size $k$, each occurrence of an substitution error results in up to $k$ erroneous $k$-mers. The erroneous $k$-mers affects the assembly process in several ways. They introduce structures such as bubbles, dead ends, and spurious links, thus affecting graph traversal and the correctness of the assembly output. In addition, they can dramatically increase the size of the observed $k$-mer set $M'$ compared to the true distinct $k$-mers in the genome. The corresponding increase in the overall size of the de Bruijn graph increases memory requirement, computation time, and communication cost.

To allow the Bruno library to effectively serve as a core component of a parallel assembler and sequence analysis tool, it needs to also support detection and removal of erroneous $k$-mers and graph structures that are introduced by these erroneous $k$-mers. We focus primarily on the detection and removal of the manifestations of short read sequencing errors, primarily substitutions, in de Bruijn graphs.

In this chapter, we describe an algorithm to perform frequency based filtering to eliminate erroneous edges and vertices from the de Bruijn graph during construction, as well as a

communication- and work-minimizing alternative. We then describe the generic process of defining, detecting, and removing erroneous graph structures. Subsequent to error removal, the graph may require re-compaction. We describe an optimized chain re-compaction algorithm with reduced communication message size and iterations.

## 5.1 Manifestation of Sequencing Errors

Erroneous $k$-mers gives rise to structures in the de Bruijn graph including spurious links, bubbles, and dead ends. Velvet [7] and ABySS [8] describe these structures. Figure 5.1 shows examples of each structure.

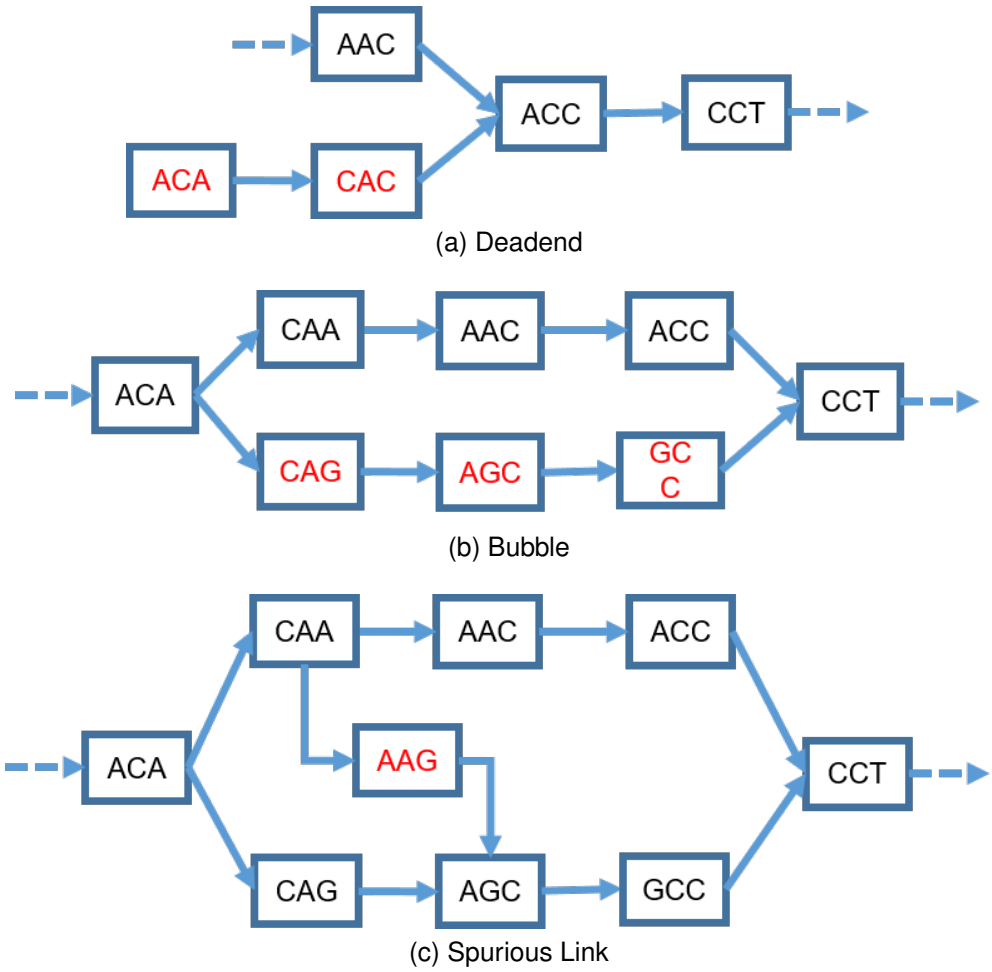



Figure 5.1: Examples of topologies that may arise due to erroneous bases introduced by the sequencer. In each case, erroneous $k$-mers are shown in red.

Dead-ends forms when an erroneous base is present in the first or the last $k$ bases of a read and the resulting $k$-mers deviates from the true genome sequence. Since the erroneous base is near the ends of a read, there may not exist $k$-mers that connect consequent erroneous path back to the true genome sequence. The $k$-mer therefore forms a chain that terminates at the last $k$-mer of the read.

Bubbles, on the other hand, consists of an alternative path with an erroneous base that does reconnect to the true genome. Bubbles can form when an erroneous base occurs in the middle $L-2k$ bases of a read. The $k$-mers flanking the erroneous base are true $k$-mers from the genome. The erroneous base therefore creates an alternative path, forming a bubble.

Spurious links can occur when a sequencing error produces a true $k$-mer by chance, or when two dead-ends merge by chance. The true paths to which the dead-ends belong are separate, and therefore spurious links erroneously connect two separate paths of the de Bruijn graph.

More complex structures can arise from the presence of erroneous $k$-mers within a graph neighborhood. Velvet [7] proposed "tour bus", a breadth first search based algorithm, in order to identify source error for complex bubble-containing structures.

We believe that such algorithm introduces complexity with limited benefits given current next generation sequencer capabilities. Read lengths have increased by up to an order of magnitude without increases in error rate. The longer reads provide longer range context for the down stream gap closing and scaffolding tasks. As sophistication in error resolution necessitate local neighborhood traversal, complex bubble resolution may be best handled as part of the gap closing and scaffolding activities.

We believe that the combination of frequency-based filtering, dead-end and simple bubble removal, and scaffolding can effectively identify and address the majority of errors in the reads and the de Bruijn graph. Additionally, the Bruno library aims to provide generic, reusable de Bruijn graph building blocks and operations. We therefore focus our efforts on frequency-based filtering and dead-ends and simple bubbles detection and removal.

## 5.2 Error Detection and Removal Strategies

Different strategies have been developed for detecting, removing, and even correcting sequencing errors in the context of genome assembly. Error correction in reads can be used as a pre-processing step for assembly. Quake [11], Reptile [12], and Musket [16] are example stand-alone read error correction tools. AllPaths-LG [53] incorporates its own pre-assembly read error correction tool. The principal approach to these error correction tools is based on $k$-mer frequency spectrum.

A second approach involves filtering the low frequency $k$-mers in the input during or immediately after graph construction. A $k$-mer is inserted if its frequency is higher than a user defined threshold. HipMer [9] uses a distributed bloom filter to remove low frequency $k$-mers. Similarly ParBiConnect [66] and BCALM 2 [59] filters $k$-mer below a threshold.

Read error correction and absolute frequency thresholding may not be sufficient in identifying and removing erroneous $k$-mers, however. Bubbles, dead-ends, and spurious links in the de Bruijn graph hint at the presence and location of erroneous $k$-mers. Coupled with application specific heuristics, such as lengths of dead-ends and bubbles and the relative edge frequencies at the branch nodes, sequences of erroneous $k$-mers in alternate paths can be identified. ABySS [8] and Velvet [7] are two assemblers that explicitly identify and correct these erroneous graph structures.

### 5.2.1 Bruno Library Error Detection and Removal

In this work we focus primarily on the detection and removal of the manifestations of short read sequencing errors, primarily substitutions, in de Bruijn graphs. We limit the scope of algorithmic investigation and implementation to this subset in order to maintain flexibility and generality of the Bruno Library. The criteria used to define erroneous graph structures is closely tied to the application requirements. Similarly, the decision to remove or to correct, as well as the heuristics for correcting errors, are application specific. The

Bruno library therefore focuses on providing only the common and necessary algorithms and functionalities in the context of error detection and removal.

The Bruno library provides functionalities for finding and removing $k$-mers with low absolute or relative frequencies. Spurious links are typically removed via $k$-mer frequency thresholds ([7],[8]). Bruno also provide the capability of detecting and removing simple bubbles and dead-ends. The application-specific definitions for erroneous graphs structures can be encoded as modular predicate functions for the Bruno API, thus improving flexibility and generality. Such definitions include the lengths of bubbles and dead-end chains, relative frequencies of the chains in a bubble, etc. Once detected, the target structures can be processed according to application requirements, for example merging bubbles or deleting dead-ends, then modify the de Bruijn graph accordingly.

In the subsequent sections, we describe the algorithms for detecting errors based on thresholds and de Bruijn graph structures. We also describe the erroneous $k$-mer and bubble and dead-end removal processes and related algorithms to illustrate the capabilities of the library. We note that while Bruno supports graphs that records the existence of edges, error detection often requires frequency information. The discussion therefore presumes that edge and $k$-mer frequencies are recorded in the de Bruijn graph.

### 5.3 Frequency-based Error Detection and Removal

As single base errors are expected to be infrequent, we also expect the resultant erroneous $k$-mers to occur with low frequency in $S$. Identifying erroneous $k$-mer by frequency is therefore a common operation for $k$-mer indices and de Bruijn graphs. As an edge in the graph $G$ connects two $k$-mers, filtering $k$-mers by frequency directly impact edge frequency, which may be used by applications during graph traversal. It is therefore important to maintain consistency of frequencies of edge and $k$-mers to the extent possible.

In Section 4.4.1 we introduced the de Bruijn graph data structure. A vertex in the graph is represented by a $k$-mer and associated metadata, namely the $k$-mer and edge ($k$+1)-mer

frequencies.

$$\langle f_{\hat{m}A}, f_{\hat{m}C}, f_{\hat{m}G}, f_{\hat{m}T}, f_{A\hat{m}}, f_{C\hat{m}}, f_{G\hat{m}}, f_{T\hat{m}}, f_{\hat{m}}\rangle$$

The frequencies are computed by accumulating the number of occurrences of each edge in the sequence data. As an edge $(v, v')$ consist of two $k$-mers with $k-1$ overlap, it corresponds to a ($k$+1)-mer in the input sequence. Similarly, two successive edges, sharing a central $k$-mer, correspond to a ($k$+2)-mer in the input.

We begin our discussion by first expanding on the notation presented in Section 4.4.1. Let $f_{q\hat{m}r}$ denote the frequency of a ($k$+2)-mer, $q\hat{m}r$, with canonical $k$-mer $\hat{m}$ at the center, and $q$ is the 5' character and $r$ is the 3' character of the ($k$+2)-mer.

The relationship between the ($k$+2)-mer, ($k$+1)-mer, and $k$-mer frequencies are:

$$f_{\hat{m}} = \sum_{q\in\{A,C,G,T,\emptyset\}} \sum_{r\in\{A,C,G,T,\emptyset\}} f_{q\hat{m}r} \tag{5.1}$$

and

$$f_{q\hat{m}} = \sum_{r\in\{A,C,G,T,\emptyset\}} f_{q\hat{m}r}, q \in \{A, C, G, T, \emptyset\} \tag{5.2}$$

$$f_{\hat{m}r} = \sum_{q\in\{A,C,G,T,\emptyset\}} f_{q\hat{m}r}, r \in \{A, C, G, T, \emptyset\} \tag{5.3}$$

Note that $\emptyset$ indicates empty character at the 5' and/or 3' character positions.

The additive nature of the frequency computation implies that given a set of ($k$+2)-mers with identical central $k$-mer, the accumulated $k$-mer and ($k$+1)-mer frequencies are consistent. However, this process also embodies a loss of context. Namely, it is not possible to determine the ($k$+2)-mer frequencies that contributed to a particular ($k$+1)-mer's frequency, nor the ($k$+1)-mer frequencies that contributed to a $k$-mer's frequency.

The consequence of this loss of context is ambiguity in the heuristics for maintaining frequency consistency when one or more edge (($k$+1)-mer) or vertex ($k$-mer) frequencies

are changed. For example, when thresholding an in-edge ($k$+1)-mer frequency, it's not possible to exactly determine the changes necessary in the out-edge ($k$+1)-mer and the $k$-mer frequencies in order to maintain consistency. Application-specific heuristics are often used, but may introduce errors during graph traversal. Figure 5.2 shows one potential scenario where consistency heuristics may be hard to devise.


Figure 5.2: An example scenario where ambiguity is encountered when devising heuristics for maintaining consistency after edge frequencies are modified. The dashed lines represent ($k$+2)-mers. The labels $x$ and $y$ represent frequencies. The vertex "G" pre-filtering has frequency $x + y$. Suppose $x$ is lower than the frequency threshold, then post-filtering "G" should have frequency $y$ in 5.2a and $x + y$ in 5.2b. However, there is no way to distinguish the ($k$+2)-mer contributions in the two cases and therefore no consistent strategy to adjust the frequency of "G" accurately.

In the Bruno library, frequency-based filtering can be accomplished in two ways. The $k$-mer and ($k$+1)-mer frequencies can be threshold by modifying the frequencies of the source ($k$+2)-mers. This approach leverages the accumulation process to ensure consistency of the final frequencies, but applies best during the graph construction process. We term this approach "bottom-up". In contrast, the "top-down" approach directly modifies the edge and $k$-mer frequencies, relying on application-level heuristics to maintain frequency consistency. While we prefer the "bottom-up" approach to ensure frequency consistency, we recognize the need for and provide the "top-down" approach for frequency-based $k$-mer filtering.

### 5.3.1 Bottom-Up Frequency-based Filtering

By modifying the frequencies of ($k$+2)-mers and selectively removing the 5' and/or 3' characters, we can effect frequency-based filtering for vertices ($k$-mers) and edges (($k$+1)-mers).

To perform bottom-up filtering, we first construct a ($k$+1)-mers count index using Kmerind [39]. The low frequency ($k$+1)-mers are deleted from the index via a local linear scan. The input $\langle k$-mer, $e\rangle$ tuples from Section 4.3.3, corresponding to a ($k$+2)-mers in the reads, are then modified based on the ($k$+1)-mers frequencies according to Algorithm 5.1.

Each $\langle k$-mer, $e\rangle$ is locally transformed into two ($k$+1)-mers, and the distributed ($k$+1)-mers count index is queried for their existence. If a ($k$+1)-mer does not exist, i.e. has been filtered, the corresponding $4$-bit group in $e$ is cleared. The existence of a ($k$+1)-mer can be represented by as few as $1$-bit during query result communication. Since the tuples are generated in order of appearance in the reads, successive tuples share a common ($k$+1)-mer. We therefore only query for the ($k$+1)-mers that are $3'$ extensions of the $k$-mers in the input tuples, and reuse the query results for the $5'$ extension ($k$+1)-mers. Once the input $\langle k$-mer, $e\rangle$ tuples are modified, they are then inserted into the de Bruijn graph as described in Section 4.3.3. The ($k$+1)-mer count index is deleted after graph construction to reduce memory usage.

**COMPLEXITY ANALYSIS**: Bottom-up filtering inserts several steps to modify $\langle k$-mer, $e\rangle$ tuples prior to graph construction. Overall two rounds of communication are required with message size $N/p$, where $N$ is the total number of ($k$+1)-mers.

Construction of the ($k$+1)-mer count index requires $O(N/p + \tau p + \mu N/p)$ time for communication and computation. Removing low frequency ($k$+1)-mers require a linear scan of the local hash table in $O(U/p)$ time, where $U$ is the number of distinct ($k$+1)-mers.

Generating the 3' ($k$+1)-mers from the ($k$+2)-mers requires $O((N+|R|)/p)$ time, where $N+|R|$ is the number of ($k$+2)-mers and $|R|$ is the number of reads. Since we include both the half ($k$+2)-mers at the 5' and 3' ends of each read, and each ($k$+2)-mer correspond to $1$

**Algorithm 5.1** Bottom-up ($k$+1)-mer frequency filtering

```
1: function FILTER_BY_FREQUENCY(P_i = ⟨k-mer, e⟩ pairs, I_i = (k+1)-mer count index)
2:     Q_i ← array of size |P_i|
                                  ▷ Extract the 3' (k+1)-mers from (k+2)-mer input
3:     for j ← 0, (|P_i| − 1) do
4:         ⟨l, m, r⟩ ← P_i[j]
5:         e ← concat(m, r)
6:
7:         Q_i.append(canonical(e))
8:     end for
                                                  ▷ Query for (k+1)-mer frequencies
9:     R_i ← I_i.exists(Q_i)
                                  ▷ Modify (k+2)-mers based on (k+1)-mer frequencies
10:    for j ← 0, (|P_i| − 1) do
11:        if R_i[j] == 0 then
12:            P_i[j].r ← ∅
13:            P_i[j + 1].l ← ∅
14:        end if
15:    end for
16:    return P_i
17: end function
```

($k$+1)-mer except for the last half ($k$+2)-mer of the read.

Querying the ($k$+1)-mer index requires $O(N/p + \tau p + \mu N/p)$ time for communication and computation, and modifying the ($k$+2)-mers again requires $O((N + |R|)/p)$ time.

### 5.3.2 Optimized Bottom-Up Frequency-based Filtering

The objective of Algorithm 5.1 is to filter out low frequency edges in the input $\langle k$-mer, $e\rangle$ pairs. It requires communication with message size $N/p$ to construct (one-way communication) and query (round trip communication) the ($k$+1)-mer count index and at least two linear scans of the input $\langle k$-mer, $e\rangle$ pairs to filter the low frequency edges prior to construction of $G$.

In this section we propose an optimized algorithm for filtering low frequency edges prior to $G$ construction that reduces the communication to a single one-way communication with message size $N/p$, local frequency calculation in $O(N/p)$ time, and filtering in

$O(U/p)$ time, where $N$ is the total number of $\langle k$-mer, $e\rangle$ pairs and $U$ is the total number of distinct $\langle k$-mer, $e\rangle$ pairs in the input data set.

We first observe that graph construction without frequency-based filtering involves file parsing to generate the $\langle k$-mer, $e\rangle$ pairs, followed by insertion into essentially a distributed hash table, with $k$-mer as key and a custom reduction operation to count the occurrences of the edges encoded in $e$. Since the frequencies are accumulated using addition, which is an associative operator, we can equivalently construct $G$ from the frequencies of distinct $\langle k$-mer, $e\rangle$ pairs (or equivalently, ($k$+2)-mers).

In other words, we can first compute the frequencies of $\langle k$-mer, $e\rangle$ pairs in a count index ${}_2I$, and then insert each element of ${}_2I$ into $G$. If we further specify the top level hash function of ${}_2I$ to assign $\langle k$-mer, $e\rangle$ pairs to processors by $k$-mer, while using the entire pair $\langle k$-mer, $e\rangle$ as key for the lower level local hash table, then all $\langle k$-mer, $e\rangle$ (($k$+2)-mers) sharing the same $k$-mer are grouped to the same processor, while the frequencies of each distinct ($k$+2)-mer is counted separately. Later insertion into $G$ are then local operation since $G$ uses $k$-mer for processor assignment.

We next leverage the relationships between ($k$+2)-mer, ($k$+1)-mer, and $k$-mer frequencies. Given a set of distinct ($k$+2)-mers with a common central $k$-mer, and their frequencies, we can compute the $k$-mer frequency as well as the 5' and 3' ($k$+1)-mer frequencies directly using Equations 5.1, 5.2, and 5.3. If the ($k$+2)-mers are assigned to processors based on the central $k$-mer, then computing $k$-mer and ($k$+2)-mer frequencies are completely local operations, and independent of any other $k$-mer or ($k$+2)-mer frequencies.

The frequency of a ($k$+1)-mer, on the other hand, are associated with both its 5' and 3' $k$-mers and therefore the corresponding ($k$+2)-mers. Let $m_{i-1}$,$m_i$, $m_{i+1}$, and $m_{i+2}$ be $4$ $k$-mers with successive $k$-1 overlaps. Then $(m_{i-1}, m_i, m_{i+1})$ and $(m_i, m_{i+1}, m_{i+2})$ are two ($k$+2)-mers sharing a ($k$+1)-mer overlap, $(m_i, m_{i+1})$. Since the *($k$+2)-mers are generated via a sliding window on the input sequence, each ($k$+1)-mer occurs twice in two successive ($k$+2)-mers.

This "double" appearance conveniently allows us to compute the ($k$+1)-mer frequencies locally from ($k$+2)-mer frequencies when they are partitioned using the central $k$-mer. One special case exists, however. When a ($k$+1)-mer is identical to its reverse complement, then its 5' $k$-mer and the reverse complement of its 3' $k$-mer are also identical. For de Bruijn graphs that uses canonical $k$-mers as keys, the ($k$+1)-mer is counted twice and the frequency value is exactly $2\times$ the true value. In this case the ($k$+1)-mer frequency must be reduced by half.

In summary, by assigning $\langle k$-mer, $e\rangle$ pairs to processor by $k$-mer, the content of count index ${}_2I$ can be locally inserted into $G$ to construct the graph. The distribution groups ($k$+2)-mers with the same central $k$-mer together, thus allowing $k$-mer and ($k$+1)-mer frequencies to be computed completely locally. Using the accumulated frequencies instead of the input $\langle k$-mer, $e\rangle$ pairs reduces the number of data elements to filter by frequency and insert into $G$ from $N/p$ to $U/p$.

The optimized bottom-up frequency-based filtering algorithm is shown in Algorithm 5.2. The $\langle k$-mer, $e\rangle$ is first inserted into the specialized count index to generate ($k$+2)-mer frequencies. The local elements of the count index are then extracted and sorted using the central $k$-mer in the comparison. Once sorted, each group of $\langle k$-mer, $e\rangle$ frequencies sharing the same $k$-mer are summed, and the result used to modified the $\langle k$-mer, $e\rangle$ pairs, which are then inserted locally into $G$.

**COMPLEXITY ANALYSIS**: The optimized algorithm invokes a single distributed count index `insert` operation, with computation and communication complexity of $O(N/p + \tau p + \mu N/p)$ to produce $U/p$ entries in the count index. The count index entries are then sorted by the central $k$-mer in $O(U/p \log(U/p))$ time. The sorted array is then processed in groups sharing the same $k$-mer via a linear scan, and inserted locally into $G$. This filtering and graph construction step requires $O(U/p)$ time.

**Algorithm 5.2** Optimized Bottom-up Frequency-based Filtering and Graph Construction

```
1: function FILTER_BY_FREQUENCY_OPT(P_i = ⟨k-mer, e⟩ pairs)
2:     _2I_i ← distributed ⟨k-mer, e⟩ count index,
3:             top level hash by k-mer,
4:             lower level hash table with ⟨k-mer, e⟩ as key.
5:     _2I_i.insert(P_i)
6:     Q_i ← local elements in _2I
7:     local sort Q_i by k-mer in ⟨k-mer, e⟩
8:     for each group of ⟨k-mer, e⟩ in Q_i with same k-mer key do
9:         compute k-mer frequency
10:        compute (k+1)-mer frequencies for in-edges and out-edges
11:
12:        Modify e in each ⟨k-mer, e⟩ in group based on k-mer, (k+1)-mer and ⟨k-mer,
       e⟩ frequencies.
13:        locally insert modified ⟨k-mer, e⟩ into G.
14:    end for
15: end function
```

### 5.3.3 Top-Down Frequency-Based Filtering

This mechanism of node and edge filtering relies directly on the predicated variants of Level 1 operations `find` and `erase` as described in Section 4.4. The user defines the desired predicate function.

Both absolute and relative threshold can be applied to the graph edges, such that edges with few occurrences, or those that have significantly lower frequencies when compared to their peer at the same vertices, can be removed. To filter graph vertices, absolute threshold can apply to $k$-mer frequencies as well.

As noted in Section 5.3, such modification of the graph can introduce inconsistencies in the vertex and edge frequencies, thus affecting subsequent graph analysis and traversal. We also note that spurious links can be identified and removed via this mechanism.

**COMPLEXITY ANALYSIS** Top-down frequency filtering applies a predicate function globally to select and/or erase edges and vertices. The selection process is applied to the local hash tables, without communication, in $O(|V|/p)$ time, where $|V|$ is the total number of vertices in the graph $G$. During removal, the vertices at both ends of each edge to be deleted

must be updated. Since each vertex has at most $2|\Sigma|$ edges, and the target vertex of an edge may reside on a different processor, distributed `erase` hash table operation from Kmerind is used, with complexity $O(|\Sigma|M/p + \tau p + \mu|\Sigma|M/p)$, where $M$ is the total number of distinct vertices affected.

## 5.4 Graph Structure-based Error Detection and Removal

Graph structures that arise due to the presence of erroneous $k$-mers include bubbles and dead-ends, both of which consist of chains with specific properties. Namely, dead-ends are chains connected to the rest of the graph at only one end, while bubbles are chains that are connected to the same branch vertices at both ends.

To detect bubbles and dead-ends, we therefore first need to find the terminal vertices of the chains. Detection algorithms for dead-ends and bubbles in ABySS [8] and Velvet [7] does so by iterating over chain vertices. As defined in Section 4.2.1, the set of chains in a de Bruijn graph $G$ is $\mathbb{C}$, with each chain consisting of a set of vertices $C \equiv \{c \in V\}$ and with length $|C|$. Here we denote the union of vertex sets for all chains in $\mathbb{C}$ as $V_c$. Velvet and ABySS therefore has computational complexity of $O(|V_c|)$.

Traversal along a chain in a distributed memory setting potentially requires communication for each vertex visited, as successive vertices may reside on different processors. A naive traversal strategy, such as that in ABySS, would require up to $O(|C|_{max})$ rounds of communication. For typical data sets the maximum chain lengths may reach $10^3$ to $10^4$ in length, rendering such a traversal pattern inefficient.

Our bubble and dead-end detection algorithms instead operates on compressed chains. As the primary criteria for deciding whether a chain is a bubble or a dead-end lies in how the terminal vertices are connected to the rest of the graph. The approach relies on two assumptions:

1. Chains in the graph has been previously compacted, for example using the algorithms described in Chapter 4.

2. The chain terminal vertices have sufficient associated metadata in themselves for evaluation of any additional application-specific predicates.

The associated metadata may include length of the chain and frequencies of edges between the termini and their 5' and 3' branch neighbors.

Under such conditions, bubble finding and dead-end detection require only examination of the chain terminal vertices with cardinality of $|\mathbb{C}|$. We note that to find the chain terminal vertices still requires scanning through all chain vertices with complexity $O(|V_c|)$. However, since each terminal vertex, represented as a $d$-path, has the identifier for its counterpart at the opposite end of the chain, A single communication round with message size $|\mathbb{C}|$ is sufficient to "traverse through the chain".

We first present a compressed chain representation that facilitates predicate evaluations on chains under the assumptions above. We also describe the algorithm for converting the ordered vertices to this compressed structure. The algorithms for detecting the bubbles and dead-ends are presented next, as well as the approach for removing them from the graph. Finally, once these structures are modified in the graph, new chains may formed. We describe an optimized algorithm for re-compacting the chains.

### 5.4.1 Compressed Chain Representation With Frequency Metadata

To detect the three types of erroneous topologies, we begin by first computing a compressed representation of the chains along with summaries of their metadata. Generation of compressed chain representation is summarizing activity, thus a Level 2 operation in the Bruno Library.

Our compressed representation, *c*-path for *chain*-path, is defined as

$$\langle {}_j\ddot{m}_{j-1}, m_j, {}_j\ddot{m}_l, {}_j\ddot{m}_{l+1}, d_{jl}, f_{qm_j}, f_{m_jr} \rangle \tag{5.4}$$

where $m_j$ is the chain representative $k$-mer as defined in Section 4.2.4. The $k$-mers $m_j$

and ${}_j\ddot{m}_l$ correspond to the two terminal vertices of a chain, while $d_{jl}$ is the length of the chain. The $k$-mer ${}_j\ddot{m}_{j-1}$ and ${}_j\ddot{m}_{l+1}$ are the 5' and 3' neighbors of $m_j$ and ${}_j\ddot{m}_l$, respectively. The values $f_{qm_j}$ and $f_{m_jr}$ correspond to the edge, or ($k$+1)-mer frequencies of the edges represented by $({}_j\ddot{m}_{j-1}, m_j)$ and $({}_j\ddot{m}_l, {}_j\ddot{m}_{l+1})$. Positive values indicate that the edges exist, while $0$ indicates that the edge does not exist, and the corresponding $k$-mer is undefined. All $k$-mers are on the same strand as as $m_j$.

The information needed to populate the *c*-paths are contained in $d$-paths of the chain terminal vertices, in chain map. Recall that $d$-paths are defined in Equation 4.2 as

$$\hat{p}_{ijl} = \langle {}_j\ddot{m}_i, d_{ij}, \hat{m}_j, d_{jl}, {}_j\ddot{m}_l \rangle$$

For terminal vertex, at least one of $d_{ij}$ and $d_{jl}$ is $0$. Suppose $d_{ij} == 0$, then $\hat{m}_j$ is the 5' terminal vertex. The $k$-mer ${}_j\ddot{m}_i$ either represents a branch vertex, or is identical to $\hat{m}_j$ if $\hat{m}_j$) does not have a 5' edge. The distance $d_{jl}$ is then length of the chain, and ${}_j\ddot{m}_l$ represent the 3' terminal vertex of the chain. The chain representative is the lexicographically smaller of $\hat{m}_j$ and the reverse complement the 3' $k$-mer $rc({}_j\ddot{m}_l)$.

Similarly, if $d_{jl} == 0$, then $\hat{m}_j$ is the 3' terminal vertex. The $k$-mer ${}_j\ddot{m}_l$ is either a branch vertex or identical to $\hat{m}_j$. The value $d_{ij}$ is the length of the chain, and ${}_j\ddot{m}_i$ is the 5' terminal vertex of the chain. The chain representative is the lexicographically smaller of the reverse complement of the central $k$-mer $rc(\hat{m}_j)$ and the 5' $k$-mer ${}_j\ddot{m}_i$.

A $d$-path therefore has sufficient information to fill $3$ of the $4$ $k$-mers in a $c$-path, specifically $m_j$ and ${}_j\ddot{m}_l$, and one of ${}_j\ddot{m}_{j-1}$ or ${}_j\ddot{m}_{l+1}$. The chain length $d_{jl}$ can be directly populated as well. The frequencies $f_{qm_j}$, $f_{m_jr}$ are present in $G$, provided that the frequency metadata for each vertex is stored in the graph, and can be retrieved via queries for the vertices $m_j$ and ${}_j\ddot{m}_l$.

The compressed chain representation generation algorithm leverages two properties. First, two terminal vertex $d$-paths of a chain yield identical chain representative $k$-mers by

definition, regardless of the distribution of the $d$-paths to the processors. This provides a way to re-order the terminal vertices, or alternatively the corresponding $c$-paths.

Second, since $G$ and $\mathbb{C}$ are stored in distributed hash tables with identical top level hash function, the $k$-mers used as hash keys are identically distributed among the processors. Querying in $G$ for edge frequencies of the central $k$-mer from a $k$-path therefore can proceed *locally* without communication.

Algorithm 5.3 outlines the process to construct the $c$-paths for all chains in $\mathbb{C}$. Subscript $i$ indicates the subset of data on processor $i$. We begin by iterating over all compacted chain vertices in $V_c$, and selecting terminal vertices with $0$ as one or both of the distance values. The selected terminal vertex $d$-path is used to partially fill a $c$-path, including the edge frequency retrieved from the local hash table in $G$.

The partially populated $c$-paths are then parallel sorted by comparing the chain representative $k$-mers $m_j$, with the objective of grouping the $c$-paths for the same chain together. A linear scan then can be used to merge two successive $c$-paths with identical $m_j$ together, forming complete $c$-paths that each represents a compressed chain.

We note that the first step in the algorithm conditionally operates on each chain vertex and therefore can be implemented using Bruno's Level 1 `find` operation coupled with a transformation. The compressed chain representation generation is itself considered a Bruno Level 2 operation.

**COMPLEXITY ANALYSIS** Identifying chain terminal vertices and converting the $d$-paths to $c$-paths require $O(|V_c|/p)$ time to scan through the input and produces $2|\mathbb{C}|/p$ number of $c$-paths, two for each chain. Since frequency value retrieval is local using a hash table, $2|\mathbb{C}|/p$ local queries each with expected $O(1)$ time for a query complexity of $O(|\mathbb{C}|/p)$. Parallel sorting using distributed memory sample sort requires $O(|\mathbb{C}|/p \log(|\mathbb{C}|/p))$ for local computation and $O(\tau p + \mu(p + |\mathbb{C}|/p))$ time for communication. The final merge requires $2|\mathbb{C}|/p$ time. The algorithm, when compared to sequential traversal of chain vertices, reduced the communication volume from $O(|V_c|/p)$ to $O(|\mathbb{C}|/p)$ and the number of

**Algorithm 5.3** Construct Compressed Chain Representations

**function** TO_COMPRESSED_CHAINS($V_c$, $G$)
  $Q_i \leftarrow$ empty array for $c$-paths
  ▷ Construct partial $c$-paths from $d$-paths
  **for** $d$-path $v \in V_c$ **do**
    **if** $v.d_{ij} == 0$ **then** ▷ 5' terminal
      $c \leftarrow newc$-path
      **if** $v.\hat{m}_j$ is chain representative **then**
        copy $k$-mers in $v$ to the first 3 $k$-mers in $c$
        $c.d_{jl} \leftarrow v.d_{jl}$
        $c.f_{qm_j} \leftarrow G$.find($v.\hat{m}_j$) in edge frequency
      **else**
        copy $k$-mers in rc($v$) to the last 3 $k$-mers in $c$
        $c.d_{jl} \leftarrow v.d_{jl}$
        $c.f_{m_jr} \leftarrow G$.find($v.\hat{m}_j$) in edge frequency
      **end if**
      append $c$ to $Q_i$
    **else if** $v.d_{ij} == 0$ **then** ▷ 3' terminal
      $c \leftarrow newc$-path
      **if** rc($v.\hat{m}_j$) is chain representative **then**
        copy $k$-mers in rc($v$) to the first 3 $k$-mers in $c$
        $c.d_{jl} \leftarrow v.d_{ij}$
        $c.f_{qm_j} \leftarrow G$.find($v.\hat{m}_j$) out edge frequency
      **else**
        copy $k$-mers in $v$ to the last 3 $k$-mers in $c$
        $c.d_{ij} \leftarrow v.d_{ij}$
        $c.f_{m_jr} \leftarrow G$.find($v.\hat{m}_j$) out edge frequency
      **end if**
      append $c$ to $Q_i$
    **end if**
  **end for**
  parallel sort $Q_i$ by chain representative $v.m_j$
  $Q'_i \leftarrow$ empty array for $c$-paths
  **for** $idx \leftarrow 0, (|Q_i| - 2)$ **do**
    **if** $Q_i[idx].m_j == Q_i[idx+1].m_j$ **then** ▷ 2 partial $c$-paths for the same chain
      $c' \leftarrow$ merge($Q_i[idx]$, $Q_i[idx+1]$)
      append $c'$ to $Q'_i$
    **end if**
  **end for**
  return $Q'_i$
**end function**

communications rounds to 1.

### 5.4.2 Dead-end Detection and Removal

Chains that are dead-ends have $c$-paths with exactly one zero-valued edge frequencies, $f_{qm_j}$ or $f_{m_jr}$. Detecting such chains requires a local linear scan of the set of $c$-paths in $O(|\mathbb{C}|/p)$ time. User specified predicates, such as a frequency threshold for the non-zero edge frequency and/or chain length, can be applied during the linear scan.

The Bruno library provide separate operations for the detection of dead-end chains and the removal of the chains. The detection operation returns the set of matching $c$-paths are returned. An application can then apply further logic or transforms to the list of dead-end chains.

Removal of dead-end chains involves severing the edge connecting the dead-end chains from the rest of the graph and potentially removal of the chain vertices from the graph $G$ and the chain map $\mathbb{C}$.

Dead-end chains can be severed via the Level 1 distributed `erase_edges` operation with the chain-to-branch edges as the parameters. A linear scan and transform of the set of dead-ends generates the required $k$-mer pairs, which are then used directly with the `erase_edges` operation.

To remove all dead-end chain vertices from $G$ and $\mathbb{C}$, a distributed hash table with all chain representatives can be created. We then perform a distributed `exists` query using the chain representative $k$-mer of each chain vertex in $V_c$. If the chain representative exists in the temporary hash table, then the corresponding chain vertex is to be removed from $\mathbb{C}$ via the local `erase` operation. The $k$-mers of the matched vertices are used as input for $G$'s Level 1 `erase_node` operation.

Dead-end detection is considered a Bruno Level 3 operation. Similar logic applies to isolated chains, where both edge frequencies in a $c$-path are zero-valued.

### 5.4.3 Bubble Detection and Removal

A bubble consists of two or more chains with non-zero edge frequencies $f_{qm_j}$ and $f_{m_jr}$ in each chain, and common first and last $k$-mers ${}_j\ddot{m}_{j-1}$ and ${}_j\ddot{m}_{l+1}$ for all chains involved. To detect the presence of bubbles, we need to reorder the compressed chain $c$-paths based on these $k$-mers.

The bubble detection algorithm proceeds in 2 steps. First the $c$-paths are parallel sorted using the following comparator for establishing a custom *less than* relationship:

$$f(x,y) = \begin{cases} true, & (x._j\ddot{m}_{j-1} < y._j\ddot{m}_{j-1}) \vee ((x._j\ddot{m}_{j-1} == y._j\ddot{m}_{j-1}) \wedge (x._j\ddot{m}_{l+1} < y._j\ddot{m}_{l+1})) \\ false, & \text{otherwise} \end{cases}$$

where $x$ and $y$ are two $c$-paths being compared.

After sorting, $c$-paths with common first and last $k$-mers ${}_j\ddot{m}_{j-1}$ and ${}_j\ddot{m}_{l+1}$ are grouped together. A linear scan through the sorted $c$-paths then identifies chains in a potential bubble. Application-specific logic can be applied to the group of chains sharing the same branch vertices. For example, the chains may additionally need similar lengths to qualify as bubbles. Bubbles formed by erroneous $k$-mers and those formed from sequence variation such as single nucleotide polymorphism (SNP) can be differentiated at this point as well by comparing the relative edge frequencies of the bubble chains.

The bubble detection operation returns the set of $c$-paths representing all chains in all bubbles. Application specific logic can then be applied to the results, for example to correct the erroneous chain by alignment and merging the paths. Removal of one or more chain in a bubble can be accomplished following the same process as described in Section 5.4.2, through the use of Bruno Level 1 `erase_edges` and `erase_nodes` operations.

The bubble detection algorithm is considered a Bruno Level 3 operation. Parallel sorting has complexity $O(|\mathbb{C}|/p \log(|\mathbb{C}|/p))$ for local computation and $O(\tau p + \mu(p + |\mathbb{C}|/p))$ time for communication. Comparison of successive $c$-paths in the sorted array to identify

chains sharing the same neighbor branch vertices requires $O(|\mathbb{C}|/p)$ time.

### 5.4.4 Graph Re-compaction

The deletion of edges and vertices corresponding to dead-ends and simple bubbles can transform a branch vertex in to a chain vertex, and disconnect a chain terminal from its branch neighbor thus forming a dead-end terminal. The change in graph topology implies that existing chains that previously was separated by a branch vertex may now be merged via a newly converted chain vertex. In this case, it is useful to re-compact the graph and chains to reflect the new topology.

We note that in the error detection and removal process, no new edges are introduced, thus no new branch vertices can form. Bruno operations only need to be concerned with the addition of new chain vertices and not their removal.

#### *Complete Graph Re-compaction*

The Bruno library provides two different re-compaction mechanisms with different applicabilities. The first directly applies the process outlined in Chapter 4. After the graph is modified, the chain map $\mathbb{C}$ content is replaced with the current set of chain nodes in $G$, and the vertices re-ordered using Algorithm 4.2.4.

The advantage of this approach is that it can be applied after most if not all types of topological changes, for example splitting a chain. Its disadvantage lies in the computational and communication performance. Each invocation after graph modification reconstructs the chain map, thus operating on $|V_c|$ chain vertices with $O(\log(|C|_{max}))$ communication rounds. If branch vertices are converted to chain vertices, $|V_c|$ increases. If chains merge, then $|C|_{max}$ potentially can increase as well.

*Hierarchical Graph Re-compaction*

For topological modifications that are applied only at branch vertices and edges between branch vertices and chain terminals, such as dead-end and bubble removal, we propose a more efficient, hierarchical re-compaction algorithm.

The optimized re-compaction algorithm is based on the observation that vertex ordering is an associative operation, and applies to the constrained scenario where the graph modification is applied after a previous chain compaction, and modifications are limited to only removal of edges between chain terminal vertices and the associated branch vertices in $G$.

**Lemma 3.** *Vertex ordering is an associative operation.*

*Proof.* Let $c_i$, $c_j$, and $c_l$ be three chain vertices. Let $d_{ij}$, $d_{jl}$, and $d_{il}$ be the distances between $(c_i, c_j)$, $(c_j, c_l)$, and $(c_i, c_l)$ pairs in the final compacted chain. Without loss of generality let $c_j$ lie on the vertex sequence from $c_i$ to $c_l$. The distances then satisfies the relationship $d_{il} = d_{ij} + d_{jl}$.

The vertex ordering algorithm (Algorithm 4.1) updates the $d$-path of $c_l$ with a new distal vertex ${}_l\ddot{m}_i$ and new distance $d_{il}$ during in each iteration. Since the new distal vertex is chosen to be vertex at exactly $d_{il}$ vertices from $c_l$, the choice of the distal can be considered a function of $d_{il}$.

As the distance calculation is based on addition, and addition is an associative operator, vertex ordering is an associative operation. □

**Corollary 3.** *Internal chain vertices can be updated using the $d$-paths of the previous chain terminal vertices.*

*Proof.* Let ${}_tC$ be a compacted chain with ordered vertices at some time $t$. Let ${}_tc_i$ and ${}_tc_l$ be terminal vertices of ${}_tC$. Let ${}_tc_j$ be an internal chain vertex with $d$-path ${}_tp_{ijl} = \langle {}_j\ddot{m}_i, d_{ij}, \hat{m}_j, d_{jl}, {}_j\ddot{m}_l \rangle$.

Without loss of generality let ${}_tC$ be extended at time $t$+1 from the terminal ${}_tc_l$ to the new terminal vertex ${}_{t+1}c_q$. The vertex ${}_tc_l$ then becomes an internal vertex with $d$-path

${}_{t+1}p_{ilq} = \langle {}_{l}\ddot{m}_i, d_{il}, \hat{m}_l, d_{lq}, {}_{l}\ddot{m}_q \rangle$. The $d$-path for the internal vertex ${}_{t+1}c_j$ is correspondingly ${}_{t+1}p_{ijq} = \langle {}_{j}\ddot{m}_i, d_{ij}, \hat{m}_j, d_{jq}, {}_{j}\ddot{m}_q \rangle$.

By Lemma 3, the vertex ordering operation is associative and ${}_{t+1}p_{ijq}$ can be computed from its $d$-path at time $t$, ${}_{t}p_{ijl}$, and the updated $d$-path of vertex $c_l$ at time $t+1$, ${}_{t+1}p_{ilq}$. □

Graph modifications are limited to the removal of edges between the chain terminal vertices and branch vertices in $G$ with two possible effects to chain terminals. First, a terminal vertex previous connected to a branch vertex may become disconnected. In this case, no topological change occurs at this terminal vertex. Alternatively, a branch vertex connected to a chain terminal vertex at time $t$ may become a chain vertex itself at time $t$+1 as some of its edges are removed. In this the chain will be extended by the former chain vertex.

Based on Corollary 3, chain internal vertices can be updated after the terminal vertices have been updated with new connectivity information, for example from a separate chain compaction and vertex ordering process involving only terminal vertices and any newly formed chain vertices. This forms the basis for our optimized algorithm shown in Algorithm 5.4.

The first step in the algorithm copies all terminal $d$-paths from $\mathbb{C}$ into an empty hash table $\mathbb{D}$ while resetting the the terminal distances. Any recently modified vertices from $G$ are inserted into $\mathbb{D}$ if they are chains. Neighbors of current branches in $G$ are then updated in $\mathbb{D}$.

The standard vertex ordering algorithm (Algorithm 4.1) is then applied to order and compact the termini in the presence of newly converted chain vertices, thus extending and merging teh chains.

Finally, the internal $d$-paths in $\mathbb{C}$ are updated with the distances and distal $k$-mers from the $d$-paths in $\mathbb{D}$, and the new $\mathbb{D}$ with new and updated vertices are merged back into $\mathbb{C}$. We note that this is enabled by the associativity of the vertex ordering operation. Additionally, the internal $d$-path updates can be delayed until after the last pass in an iterative error

**Algorithm 5.4** Re-compact the chains in a modified graph

1: **function** RE_COMPACT($\mathbb{C}$, $G$, modified_vertices_in_$G$)
▷ Extract terminal vertices from compacted chains
2: $\mathbb{D} \leftarrow$ terminal vertices in $\mathbb{C}$
3: **for** $d$-path $c \in \mathbb{D}$ **do**
4: reset $c$ distances: if 0, reset to 1.
5: else, mark as pointing to internal node
6: **end for**
▷ Check modified vertices and insert in $\mathbb{D}$ if chain
7: **for** $v \in$ modified_vertices_in_$G$ **do**
8: **if** $v$ is a chain vertex **then**
9: transform $v$ into a $d$-path and insert in $\mathbb{D}$
10: **end if**
11: **end for**
▷ Mark neighbors of branch vertics as terminal (Section 4.3.4)
12: $Q_i \leftarrow$ empty array of $k$-mers
13: **for** $d$-path $c \in \mathbb{D}$ **do**
14: append distal $k$-mer of $c$ in $Q_i$.
15: **end for**
16: $P_i \leftarrow G$.find($Q_i$)
17: **for** vertex $v \in P_i$ **do**
18: **if** neighbor of $v$ is in $\mathbb{D}$ **then**
19: Update $\mathbb{D}$ to mark matching $d$-paths as terminal.
20: **end if**
21: **end for**
▷ Re-compact the previous terminal vertices
22: Invoke Algorithm 4.1 with $\mathbb{D}$
▷ Update previous internal vertices
23: $Q_i \leftarrow$ empty array of $k$-mers
24: **for** $d$-path $c \in \mathbb{C}$ **do**
25: append $c$'s distal $k$-mers to $Q_i$.
26: **end for**
27: Hash table $R_i \leftarrow \mathbb{D}$.find($Q_i$)
28: **for** $d$-path $c \in \mathbb{C}$ **do**
29: update $c$ with $d$-paths in $R_i$.
30: **end for**
31: merge $\mathbb{D}$ into $\mathbb{C}$, overwriting.
32: return $\mathbb{C}$
33: **end function**

removal process, as long as all terminal vertices from the initial $d$-path set $\mathbb{C}$ are updated during each invocation of Algorithm 4.1 within Algorithm 5.4.

One adjustment that is necessary for Algorithm 4.1 relates to cycle detection and thus

the stopping criteria for the iterative vertex ordering algorithm. Cycles can occur after dead-end and bubble removal.

In Section 4.2.6, potential cycle nodes are identified as having values $2^t$ during iteration $t$ for both $d$-path distances $d_{ij}$ and $d_{jl}$. As the distances in $\mathbb{D}$ are the chain lengths from a previous compaction, this condition does not generally hold during re-compaction. The core parallel list ranking algorithm (Algorithm 4.1) is agnostic of variations in vertex lengths and therefore can be re-used.

We adopted an alternative stopping criteria for re-compaction: re-compaction stops when the counts of *semi-finished* $d$-paths in $\mathbb{D}$ becomes zero, at which point all chain vertices are marked as *finished* while the cycle $d$-paths remain as *unfinished*.

**COMPLEXITY ANALYSIS** The re-compaction algorithm reduces run time in two ways. First, the iterative vertex ordering algorithm, Algorithm 4.1, which employs multiple round of communication, uses a reduced message size, from $|V_c|/p$ in the full compaction to the number of terminal vertices plus up to 2 newly formed chain vertices per chain, totaling $4 * |\mathbb{C}|$. Second, the number of communication rounds in Algorithm 4.1 is reduced from a function of the maximum chain length after error removal pass $t$, $O(\log(|_tC|_{max}))$, to the maximum number of chains to be merged, which is function of the overall graph topology.

The first step of Algorithm 5.4 locally extracts the existing terminal $d$-paths from $\mathbb{C}$ in linear time $O(|V_c|/p)$. Modified vertices in $G$ are identified by $k$-mers from the dead-ends and bubbles. At most $4$ vertices are modified per chain. The $k$-mers are distributed to the processors based on $G$'s hash function in $O(|\mathbb{C}|/p + \tau p + \mu|\mathbb{C}|/p$ time. The corresponding vertices are extracted from $G$ locally in $O(|\mathbb{C}|/p)$ time and inserted into $\mathbb{D}$. Let the set of $d$-paths in $\mathbb{D}$ be $V_d$, which is $o(|\mathbb{C}|)$.

To mark the branch neighbors as terminal, the distal $k$-mers, which represent potential branch vertices in $G$, are used to query and retrieve the vertices, and the results used to update the $d$-path distances in $\mathbb{D}$. The distal $k$-mer extraction and later terminal updates require $O(|V_d|)$ time, while communication and query require $O(|V_d|/p + \tau p + \mu|V_d|/p)$

time.

Algorithm 4.1 invocation uses requires rounds of communication equal to the maximum number of chains that at merge into a new chain, each iteration operating on at most $O(|V_d|/p)$ $d$-paths.

The final update requires a distributed query into $\mathbb{D}$ using the distal $k$-mers of the internal $d$-paths in $\mathbb{C}$, and therefore has complexity equal to $O(|V_c|/p + \tau p + \mu|V_c|/p)$.

Overall, only the first step, where the terminal $d$-paths are extracted, and the last step, where internal $d$-paths are updated, require processing all $|V_c|$ chain vertices. These steps only need to be executed once for a multi-pass error removal process. The remaining steps all operate with the terminal vertices and their (formerly) branch neighbors, thus have data size of $|V_d| \approx |\mathbb{C}|$, approximately equal to the number of initial chains. For the case where the only edges modified are between chain terminal vertices associated branches, Algorithm 5.4 is significantly more efficient in run-time complexity compared to the full compaction algorithm.

## 5.5 Summary

In this chapter we described the error correction algorithms that identifies and removes erroneous structures based on frequency or local graph structures such as bubbles and dead-ends. We presented an optimized algorithm to filter ($k$+1)-mers based on frequency during graph construction that requires no additional communication rounds over graph construction, and performs the filtering in time proportional to the number of distinct rather than all ($k$+2)-mers.

For graph-structure based error removal, we presented an algorithmic and library framework that supports user-defined predicates. The predicates are applied to the compressed chain representations of previously compacted chains to identify the chains that are potentially erroneous. We showed how such a framework supports the detection of dead-ends and bubbles.

We also presented an optimized algorithm for re-compacting of chains post error removal. The algorithm has run time proportional to the number of chains rather than the number of chain vertices in the graph, and requires rounds of communication proportional to the maximum number of chains merged. Iterative application of the erroneous structure detection and removal provide a mechanism for identifying and handling erroneous structures in the graph.

# CHAPTER 6

# DE BRUIJN GRAPH PERFORMANCE EVALUATION

In this chapter, we evaluate the work presented in Chapters 4 and 5. We assessed the run-time performance of our algorithm in three aspects: (1) effects of parameters, (2) detailed performance characterization, and (3) parallel scalability and comparison with state-of-the-art existing tools in distributed and shared memory environments. In addition, we examine the effects of input parameters, including value of $k$ and frequency thresholds. While such parameters are ultimately chosen based on application requirements and target data characteristics, choices within allowable value ranges can have a significant impact on computational performance and/or output quality.

In the following sections we evaluate the de Bruijn graph construction, compaction, and error removal functionalities. Each section includes, as appropriate, performance characterization, parameter studies and scalability results in distributed and shared memory environments. We then compare the performance of Bruno's construction and chain compaction to existing tools in both distributed and shared memory settings. Finally, we evaluate the quality of the generated unitigs, including effects of error removal with different parameters.

## 6.1 Experimental Configuration

The data sets used for our evaluations are detailed in Table 6.1. All data sets contain NGS paired end reads. Data set A is a subset of the Iowa cornfield soil metagenome data set from the Joint Genome Institute project 402461. Data set B corresponds to reads associated with human chromosome 14, and is obtained from the test data sets for HipMer. Data set E was used by AllPaths-LG [53], and consists of sequencing output of two experiments in project SRP03680 that led to data set D. Data set F, G, H, and I is from the GAGE project [93].

Table 6.1: Data sets used for experiments. All data sets contain paired end reads in FASTQ format, and all reads are of DNA sequences.

| Id | Accession | Organism | Individual | Sequencer | No. Reads | Read Len | Size (GB) |
|---|---|---|---|---|---|---|---|
| A | SRP081657 | Iowa corn field soil metagenome subset | | Genome Analyzer II/IIx | 132,783,552 | 76 - 114 | 30.4 |
| B | SRP003680 | *H. sapien* | CEU HapMap NA12878 Chr 14 | HiSeq 2000 | 62,220,801 | 101 | 10.0 |
| C | SRR2842672 | *H. sapien* | CHM1htert cell line | HiSeq 2500 | 462,468,962 | 125 | 155.3 |
| D | SRP003680 | *H. sapien* | CEU HapMap NA12878 | HiSeq 2000 | 2,873,524,171 | 101 | 694.8 |
| E | SRX027713 SRX027583 | *H. sapien* | CEU HapMap NA12878 subset | HiSeq 2000 | 1,591,644,480 | 101 | 311.0 |
| F | SRR022868 | *S. aureus* | GAGE | Genome Analyzer II | 1,294,104 | 101 | 0.29 |
| | SRR022865 | | | | 3,494,070 | 37 | 0.35 |
| G | SRR081522 | *R. sphaeroides* | GAGE | Genome Analyzer II | 2,050,868 | 101 | 0.47 |
| | SRR034528 | | | | 2,050,868 | | 0.46 |
| H | SRP003680 | *H. sapiens* | GAGE, NA12878 | HiSeq 2000 | 36,504,800 | 101 | 8.3 |
| | | | | | 222,669,408 | | 5.2 |
| I | Keck Center Biotech. Center Univ. Illinois | *B. impatiens* | GAGE | Genome Analyzer IIx | 2,405,064 | 76-101 | 0.52 |
| | | | | | 303,118,594 | 124 | 98.3 |
| | | | | | 129,118,270 | | 42.5 |
| | | | | | 65,081,280 | | 21.1 |

Systems used in our experiments are detailed in Table 6.2. Parameter study and performance characterization experiments were conducted on CyEnce at Iowa State University and Swarm at Georgia Institute of Technology. Distributed-memory experiments were conducted on Edison , a Cray XC30 supercomputer at the National Energy Research Scientific Computing Center (NERSC), and Swarm . Shared-memory experiments were performed on CompBio at the Georgia Institute of Technology using MPI with 1 core dedicated to each process.

Table 6.2: Systems used for performance evaluations and comparisons.

| System | CompBio | CyEnce | Swarm | Edison |
|---|---|---|---|---|
| CPU | Xeon E7-8870 | Xeon E5-2650 | Xeon E5-2680 v4 | Ivy Bridge |
| Cores | 4× 18 | 2× 8 | 2× 14 | 2× 12 |
| Memory | 1TB DDR4 | 128GB DDR3 | 256GB DDR4 | 64GB DDR3 |
| Network | (Shared Mem) | QDR InfiniBand | FDR Infiniband | Cray Aries |
| Storage | RAID 5 | Lustre, 8 OST | GPFS | Lustre, 24 OST |
| Compiler | GCC 5.3 | GCC 5.2 | GCC 4.9.4 | GCC 6.1 |
| MPI | OpenMPI 1.10.2 | MVAPICH 2.1.7 | MVAPICH 2.3*b* | Cray MPICH 7.4.1 |

Each experiment was repeated at least three times. The maximum wall-times from all processors were collected, and the minimum times amongst the repeats are reported as they more closely represent the capabilities of the systems. Where applicable, input and output files were striped across all available Lustre OSTs with default block size (typically 1 MB). Our code is compiled using compiler flags `-O3 -march=native` in order to enable maximum SIMD support.

## 6.2 de Bruijn Graph Construction

We begin with evaluation of the de Bruijn graph construction process. In this section, we examine the effects of $k$-value on construction speed and scalability. We also evaluate the impact of the bottom-up frequency-based error filtering as it is applied during the construction phase. We then evaluate the performance and scalability of our communication and

computation minimizing construction algorithm. For all experiments, the initial file reading times were excluded in order to exclude any file system noise and changes in behavior over time.

### 6.2.1 Parameter Studies

We evaluated the impact of varying parameters including $k$-mer length and frequency threshold on graph construction performance.

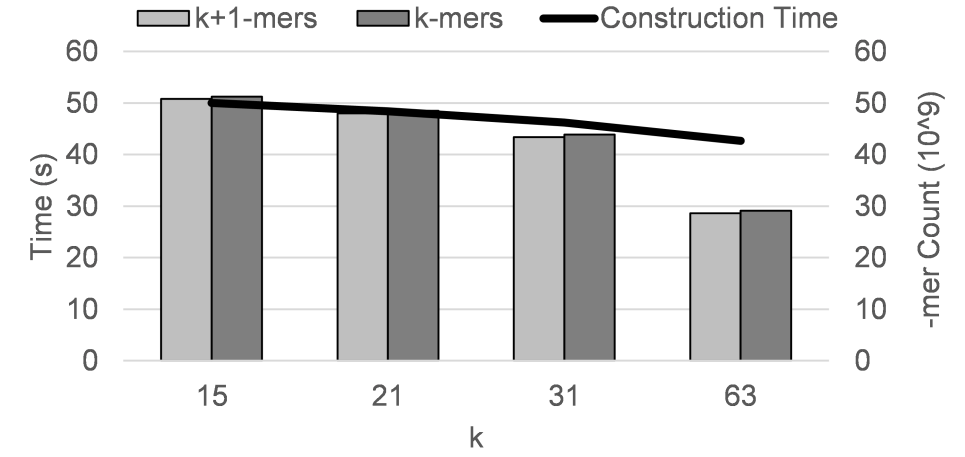
*$k$-mer Length*

Table 6.3: Total number, in *billions*, of $k$-mers, $(k+1)$-mers, and resulting graph vertices and chain vertices, and the number of chains identified in data set E for $k \in \{15, 21, 31, 63\}$.

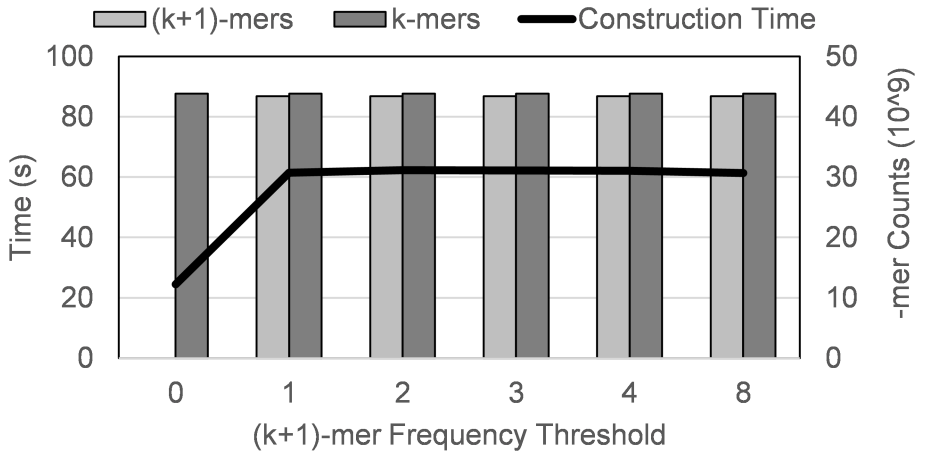
| $k$ | $k$-mers | ($k$+1)-mers | total vertices |
|---|---|---|---|
| 15 | 51.25 | 50.79 | 0.38 |
| 21 | 48.48 | 48.02 | 2.32 |
| 31 | 43.86 | 43.40 | 2.55 |
| 63 | 29.09 | 28.62 | 2.74 |

We used $64$ CyEnce nodes with a total of $1024$ cores, data set E, and $f = 2$ for $k \in \{15, 21, 31, 63\}$. We expect the number of distinct $k$-mers to increase with $k$ with an upper bound of $\max(|D|, 4^k)$, where $|D|$ is the target genome size. As $k$ increases, the number of $k$-mers parsed from each read, $L - k + 1$, where $L$ is the length of a read, is expected to decreases linearly. Similarly for ($k$+1)-mers. Table 6.3 summarizes the $k$-mer, $(k+1)$-mer, and graph vertex for data set E, which reflect these expectations.

The number of $k$-mers and $(k+1)$-mers directly impact the performance of graph construction. Figure 6.1a shows that the construction time decreased from $50.0$ seconds to $42.7$ seconds as $k$ increases from $15$ to $63$, due to decreases in the number of $k$-mers and $(k+1)$-mers. This is despite the fact that $63$-mers require two $64$ bit machine words with concordant higher costs of representation, hashing, and comparisons when compared to $15$-, $21$-, or $31$-mers.




(a) Varying $k$, Construction




(b) Varying frequency threshold $f$, Construction

Figure 6.1: Effects of varying $k$ and filter frequency on the performance of graph construction for data set E on $1024$ cores. (6.1a): The line shows the time for construction while the bars show the number of $k$-mers and $k+1$-mers, varying $k$ and setting $f = 2$. (6.5a): The line shows the time for construction while the bars show the number of chain vertices in the graph, varying $f$ and setting $k = 31$. (6.5b): the line and bars depict the same entities as 6.5a.

*Frequency Filtering Thresholds*

We next examine the impact of varying the frequency threshold for excluding erroneous edges as described in Section 5.3.1. We note that edges in the graph correspond to $(k+1)$-mers in the read set. Thus erroneous vertices and edges are excluded as a consequence of filtering the $(k+1)$-mers. This study was conducted using $64$ CyEnce nodes with a total

of $1024$ cores, data set E, $k = 31$, and frequency thresholds $f \in \{0, 1, 2, 3, 4, 8\}$. Note that for $f \in \{0, 1\}$ effectively all $(k + 1)$-mers are kept, and for $f = 0$ the filtering process is bypassed.

Table 6.4: Total number, in *billions*, of filtered $k$-mers and $(k + 1)$-mers, and the resulting graph vertices and chain vertices, and the number of chains identified in data set E for frequency threshold $f \in \{1, 2, 3, 4, 8\}$.

| $f$ | filtered distinct ($k$+1)-mers | filtered $k$-mers | vertices |
|---|---|---|---|
| 1 | 5.70 | 43.86 | 5.63 |
| 2 | 2.57 | 40.77 | 2.55 |
| 3 | 2.51 | 40.66 | 2.49 |
| 4 | 2.47 | 40.56 | 2.46 |
| 8 | 2.20 | 39.01 | 2.20 |

Table 6.4 summarizes the effects of varying frequency threshold for erroneous $(k + 1)$-mer filtering. *Filtered distinct* $(k + 1)$*-mer* refers to distinct $(k + 1)$-mers with frequency greater than or equal to the specified threshold, whereas *filtered* $k$*-mers* refer to all $k$-mers from the read set that are prefixes or suffixes in filtered $(k + 1)$-mers. The number of $(k + 1)$-mers, distinct $(k + 1)$-mers, and $k$-mers for data set E and $k = 31$ are $43.40$, $5.70$, and $43.86$ billion, respectively. As expected, increasing $f$ decreases the number of filtered distinct $(k + 1)$-mers, filtered $k$-mer, and consequently graph vertices.

Since the runtime and complexity of frequency filtering and graph construction are dominated by communication and processing of the unfiltered $(k + 1)$-mers and $k$-mers, they are not significantly affected by the frequency threshold as the graph construction time with frequency filtering, as shown in Figure 6.1a. The construction time remained at approximately $61$ seconds for $f > 0$. We note, however, that the filtering process imposes a significant overhead such that bypassing filtering reduced graph construction time to $24.5$ seconds.

### 6.2.2 Optimized Frequency Filtering

In Section 5.3.2, we proposed an optimized algorithm for bottom-up frequency-based error removal. The algorithm minimizes communication as well as reducing the data size during the actual filtering process. We compare the performance of the naïve algorithm (Algorithm 5.1) with the optimized algorithm (Algorithm 5.2).

We constructed the de Bruijn graph for data set H with $k = 31$ and frequency threshold $f = 4$ on the CompBio and Swarm systems. On CompBio , we used from $4$, $8$, $16$, $32$, and $64$ cores. The MPI implementation used memory copy for data movement. We then repeated the experiment on Swarm with $2$, $4$, $8$, $16$ and $32$ nodes with $16$ cores per nodes. Here the MPI implementation used FDR Infiniband for communication. We also processed data set I using $8$, $16$ and $32$ Swarm nodes.

Table 6.5: Performance of optimized bottom-up frequency-based filtering on construction time. The experiments were conducted using $k = 31$, $f = 4$ on data set H

| CompBio - data set H | | | | Swarm - data set H | | | | Swarm - data set I | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| cores | naïve | optimized | speedup | cores | naïve | optimized | speedup | cores | naïve | optimized | speed up |
| 4 | 552.00 | 342.66 | 1.61 | 32 | 79.99 | 46.92 | 1.70 | | | | |
| 8 | 282.20 | 172.12 | 1.64 | 64 | 45.19 | 25.88 | 1.75 | | | | |
| 16 | 151.68 | 92.76 | 1.64 | 128 | 22.93 | 12.98 | 1.77 | 128 | 216.83 | 116.06 | 1.87 |
| 32 | 80.30 | 48.19 | 1.67 | 256 | 11.96 | 6.43 | 1.86 | 256 | 110.05 | 58.43 | 1.88 |
| 64 | 48.68 | 28.72 | 1.69 | 512 | 7.12 | 3.23 | 2.21 | 512 | 58.91 | 30.44 | 1.94 |

Table 6.5 shows that on a shared memory system such as CompBio , Algorithm 5.2, labeled "optimized" consistently outperformed the naïve filtering algorithm by $61$ to $69\%$. The speed up is not strongly dependent on the core count in this case, suggesting that the primary contribution to the speed up in the share memory case is the reduction in data elements processed during filtering.

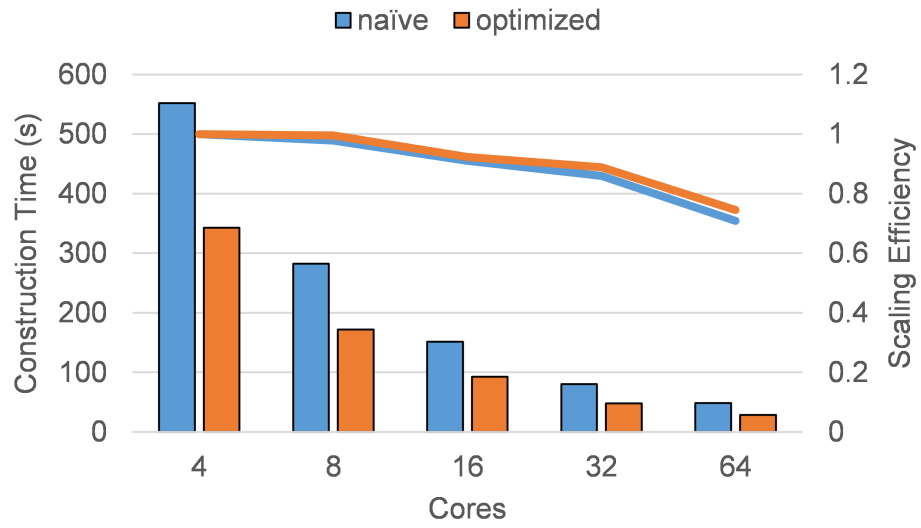
In distributed memory environment with data set H, we observed a more significant speed up, between $70$ and $121\%$. For lower core counts, the speed up was relatively constant at just above $70\%$. With $256$ and $512$ cores, we reached $86\%$ and $121\%$ improvement respectively, which may be the result of improved cache utilization with smaller per-processor element count, and/or avoidance of communication with higher latency at larger process count $p$. For data set I, the optimized algorithm compared to the naïve algorithm showed between $87$ and $94\%$ improvement.

### 6.2.3 Parallel Scalability

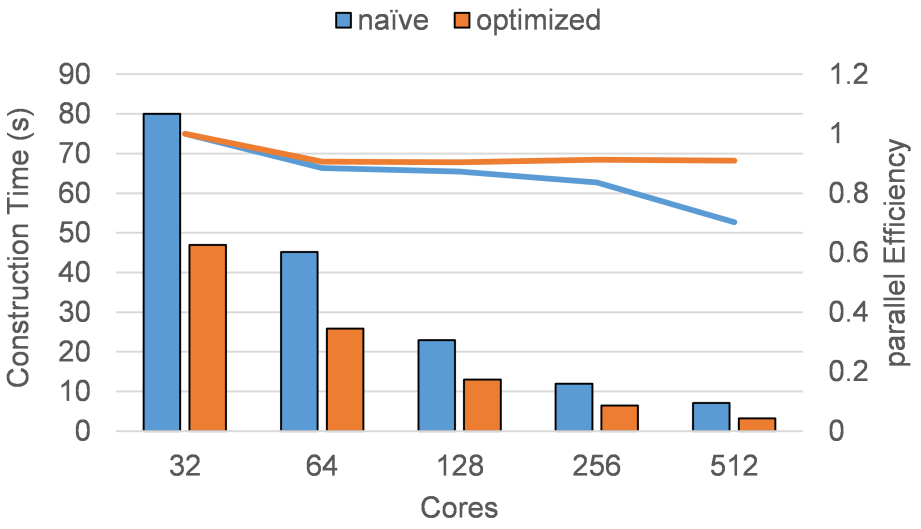
The experiments using data set H from Section 6.2.2 were also used for evaluating the scalability of the graph construction algorithm with either the naïve or the optimized frequency filtering algorithms.

Figure 6.2 shows that graph construction with the naïve bottom-up frequency filtering scaled with $70\%$ efficiency when the core count increased $16$-fold from $4$ to $64$ on CompBio and from $32$ to $512$ on Swarm . The parallel efficiency with the optimized filtering algorithm improved only slightly than the naïve version on CompBio , as the communication takes place in memory. On Swarm , on the other hand, the optimized version performance significantly better than the naïve algorithm, maintaining over $90\%$ efficiency. As stated in Section 6.2.2, this is likely the combined result of cache efficiency and reduced communication.

The results shows that the optimized bottom-up frequency-based filtering performs significantly better and can scale better than the naïve approach. Subsequent experiments,




(a) CompBio , Shared memory




(b) Swarm , Distributed memory

Figure 6.2: Scalability of graph construction algorithm, using the naïve and optimized bottom-up frequency-based filtering algorithms during construction time. The experiments were conducted using $k = 31$, $f = 4$ on data set H. Bars depict construction and filtering time (primary axis). Lines show parallel efficiency (secondary axis)

unless otherwise noted, used the optimized frequency filtering algorithm.

## 6.3 Chain Compaction

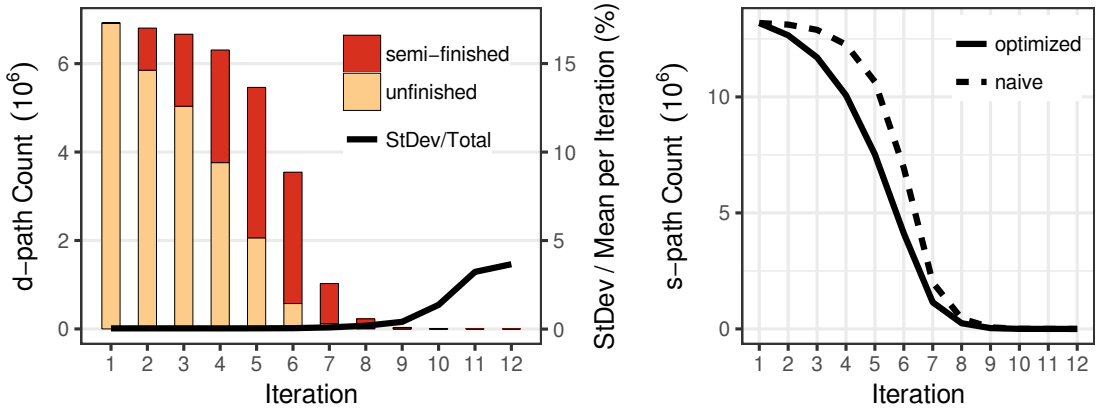
We next evaluate the de Bruijn graph compaction process. We begin by characterizing the dynamic behavior of our iterative chain compaction algorithm as it iteratively orders the chain vertices. We next evaluate the performance of our communication minimizing compaction algorithm. We then study the impact of $k$ and $f$ values on the performance of our algorithm, and finally the scaling behavior of the compaction algorithm.

### 6.3.1 Performance Characterization

We first assess the progression of the chain compaction algorithm. For these experiments, $8$ CyEnce nodes were used for a total of $128$ cores. We specified data set A as input, $k = 31$, and frequency threshold of $f = 2$. We include only time for chain compaction in this analysis.

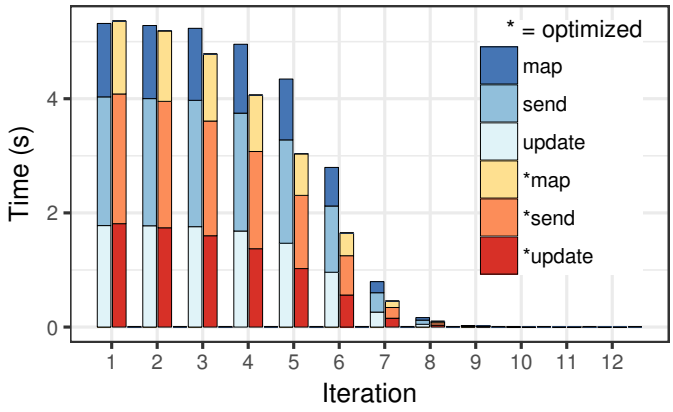
As described in Section 4.2, the chain compaction process has $3$ primary steps: *chain vertex extraction*, iterative *vertex ordering*, and collective *unitig generation* from chains. The number of iterations used by *vertex ordering* is a function of the length of the longest chain, while the performance of each iteration is determined by the number of unfinished $d$-paths during that iteration.

Figure 6.3a shows the average number of *semi-finished* $d$-paths and *unfinished* $d$-paths per core at the start of each *vertex ordering* iteration. As the iterations progress, $d$-paths are marked as *finished* as chains are compacted. We note that the rate of change depends on the distribution of chain lengths in the data set, in this case a significant number of $d$-paths became *finished* during iterations $4$ through $7$. During iterations $9$ through $12$, only a small number of *active* $d$-paths remain. The communication and computation are balanced, as the standard deviation of the unfinished $d$-path counts represents less than $0.05\%$ of the *active* $d$-paths in each of the first $6$ iterations, less than $0.5\%$ in the first $9$ iterations, and less than $4.4\%$ in all $12$ iterations. The increased load imbalance during later iterations does not




(a) $d$-path Count

(b) $s$-path Count




(c) Iteration Times

Figure 6.3: Detailed characterization of the chain compaction algorithm using data set A, $k = 31$ and $f = 2$ on 128 cores. (6.3a): The average number of *semi-finished* chain $d$-paths and *unfinished* $d$-paths per core during each iteration of *vertex ordering*. The line shows the standard deviation of the number of *active* $d$-paths as a percentage of the average *active* $d$-path count per core. (6.3b): Reduction of number of updates through selective exclusion of unproductive $s$-paths. (6.3c): Improvements in *vertex ordering* execution time per iteration. The "map" step rearranges data elements, the "send" step communicates the data to the remote processes, while the "update" step performs the $d$-path updates locally.

significantly influence the total run time as the number of remaining *active* $d$-paths is small compared to the initial counts, in this case between $4$ and $5$ orders of magnitude smaller in iterations $10$ through $12$.

### 6.3.2 Communication Reduction in Compaction

The number of $s$-paths correlate directly to the number of *active* $d$-paths. In Section 4.3.6 we outlined an approach to reduce communication volume during *vertex ordering*, by selectively constructing $s$-paths only when they produce meaningful remote updates. In this section we examine the effect of this optimization. We compare the performance of *vertex ordering* iterations with and without optimization using data set A, $k = 31$, and $f = 2$, on $8$ CyEnce nodes with a total of $128$ cores.

Figure 6.3b shows that the optimization reduced the number of $s$-paths per core. The maximum reduction occurred during iteration $5$ at $3.1$ million or $29.4\%$, and at least $1$ million during iterations $3$ through $6$. The reduction in the number of $s$-paths impacts both the communication volume as well as the computation required to perform remote updates. Figure 6.3c shows the time required to assign (`map()`), distribute (`send()`) the $s$-paths to processors, and to modify the chain $d$-paths based on received $s$-paths (`update()`). The `send()` step, representing communication, required the largest amount of time in each iteration, closely followed by the time for `update()`. In each iteration, the time used for each step is reduced significantly, with the largest reduction of $1.84$ seconds or $29.8\%$ during iteration $5$.

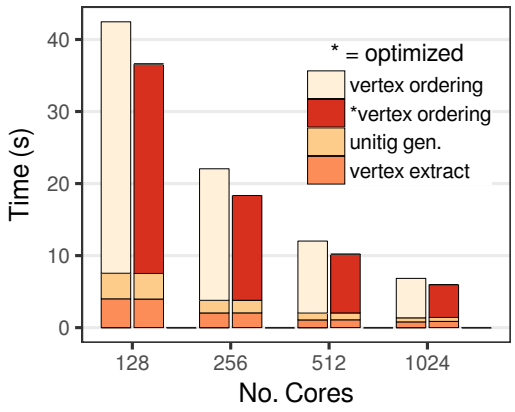



Figure 6.4: Chain compaction execution time in seconds on $128$, $256$, $512$, $1024$ cores, comparing the naïve and optimized implementations. The three steps of chain compaction are separately depicted. Data set A was used with $k = 31$ and $f = 2$.

The effect of this optimization on communication and consequently total running time is assessed by extending the experiment from $8$ to $16$, $32$, and $64$ CyEnce nodes, with total of $256$, $512$, and $1024$ cores. Figure 6.4 shows the execution times for the chain vertex extraction, ordering, and unitig generation steps as described in Section 4.3. While the chain vertex extraction and output times remained unchanged, the iterative vertex ordering times were reduced by $16.8\%$, $20.3\%$, $18.5\%$, and $16.7\%$ for $128$, $256$, $512$, and $1024$ cores respectively. Based on these results, all subsequent experiments employed this optimization.

### 6.3.3 Parameter Studies

As with graph construction, We evaluated the impact of parameters including $k$-mer length and frequency threshold $f$ on chain compaction performance. The effects of $k$ on the number of total and distinct $k$-mers as well as the number of graph vertices on data set E are summarized in Table 6.3. Similarly Table 6.4 shows the number of $k$-mers and ($k$+1)-mers post-filtering.

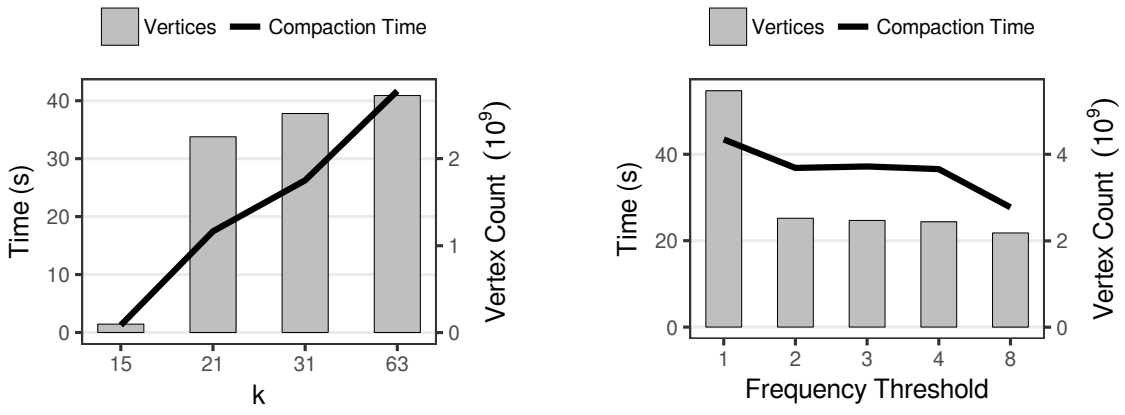
#### *$k$-mer Length*

We used $64$ CyEnce nodes with a total of $1024$ cores, data set E, and $f = 2$ for $k \in \{15, 21, 31, 63\}$. As graph vertices represent $k$-mers and edges $(k+1)$-mers, and the probability of multiple $(k+1)$-mers sharing the same $k$-prefix (or suffix) decreases with increasing $k$, we expect the number of chain vertices to increase, the number of branch vertices to decrease, and the lengths of the chains to increase while the number of chains to decrease. Table 6.6 complements Table 6.3with the numbers of chain vertices and total chain counts as $k$ is varied for data set E.

The performance of chain compaction is influenced by both the total number of chain vertices (Table 6.3) and the number of iterations $t$, which is related to the maximum chain length by $t = \lceil \log(|C|_{max}) \rceil$. Chain compaction of the de Bruijn graph for data set E required $6$, $12$, $15$, and $15$ iterations for $k = 15$, $21$, $31$, $63$, indicating that the length of the

Table 6.6: Total number, in *billions*, of graph vertices and chain vertices, and the number of chains identified in data set E for $k \in \{15, 21, 31, 63\}$.

| $k$ | total vertices | chain vertices | chains |
|---|---|---|---|
| 15 | 0.38 | 0.10 | 0.060 |
| 21 | 2.32 | 2.25 | 0.087 |
| 31 | 2.55 | 2.52 | 0.043 |
| 63 | 2.74 | 2.73 | 0.019 |




(a) Varying $k$ (b) Varying $f$

Figure 6.5: Effects of varying $k$ and filter frequency on the performance of chain compaction for data set E on $1024$ cores. (6.5a): The compaction time (line) and chain vertex count (bars) for varying $k$ and fixed filter frequency $f = 2$. (6.5b): The compaction time (line) and chain vertex count (bars) for varying frequency thresholds and fixed $k = 31$.

longest chain increased with $k$. As a consequence, the chain compaction times increased with $k$, from $1.26$ seconds for $k = 15$ to $41.7$ seconds for $k = 63$, as shown in Figure 6.5.

*Frequency Filtering Thresholds*

We next examined the impact of varying the frequency threshold on computational performance. This study was conducted using $64$ CyEnce nodes with a total of $1024$ cores, data set E, $k = 31$, and frequency thresholds $f \in \{1, 2, 3, 4, 8\}$. Note that for $f = 1$ effectively all $(k + 1)$-mers are kept.

Table 6.7 complements Table 6.4 with the numbers of chain vertices and chains in the graph for each of the tested $f$ values. The number of chains decreased as $f$ is increased to

Table 6.7: Total number, in *billions*, of graph vertices and chain vertices, and the number of chains identified in data set E for frequency threshold $f \in \{1, 2, 3, 4, 8\}$.

| $f$ | vertices | chain vertices | chains |
|---|---|---|---|
| 1 | 5.63 | 5.47 | 0.240 |
| 2 | 2.55 | 2.52 | 0.043 |
| 3 | 2.49 | 2.47 | 0.033 |
| 4 | 2.46 | 2.44 | 0.030 |
| 8 | 2.20 | 2.18 | 0.033 |

$4$, but increased at $f = 8$, likely due to linear chains being broken by the removal of lower frequency edges.

Chain compaction performance improves as $f$ increases, as shown in Figure 6.5b. The time for compaction decreased from $44.4$ seconds for $f = 1$ to $36.8$ seconds for $f = 2$ as the number of chain vertices more than halved from $5.47$ to $2.52$ billion, even while the iteration count increased from $11$ to $15$. Frequency thresholds of $2$, $3$, $4$ did not significantly impact compaction time, as they shared similar number of chain vertices and identical iteration counts. Thus for data set E, a significant portion of the erroneous $(k + 1)$-mer have cardinality of $1$. At $f = 8$, the compaction time further reduced to $27.7$ seconds as both the chain $d$-path count reduced significantly and the iteration count reduced to $14$, again likely due to chain breaking.

### 6.3.4 Parallel Scalability

We evaluated the scalability of the chain compaction and unitig generation process in both shared memory and distributed memory environments. Data sets H was used with parameters $k = 31$ and $f = 4$ on CompBio , using $4$, $8$, $16$, $32$, and $64$ cores. Data set H was compacted with the same parameters on Swarm using $2$, $4$, $8$, $16$, and $32$ nodes and $16$ of $28$ cores on each node.

Figure 6.6 shows that the graph compaction process with communication optimized compaction algorithm scaled well in distributed memory environment and relatively well in

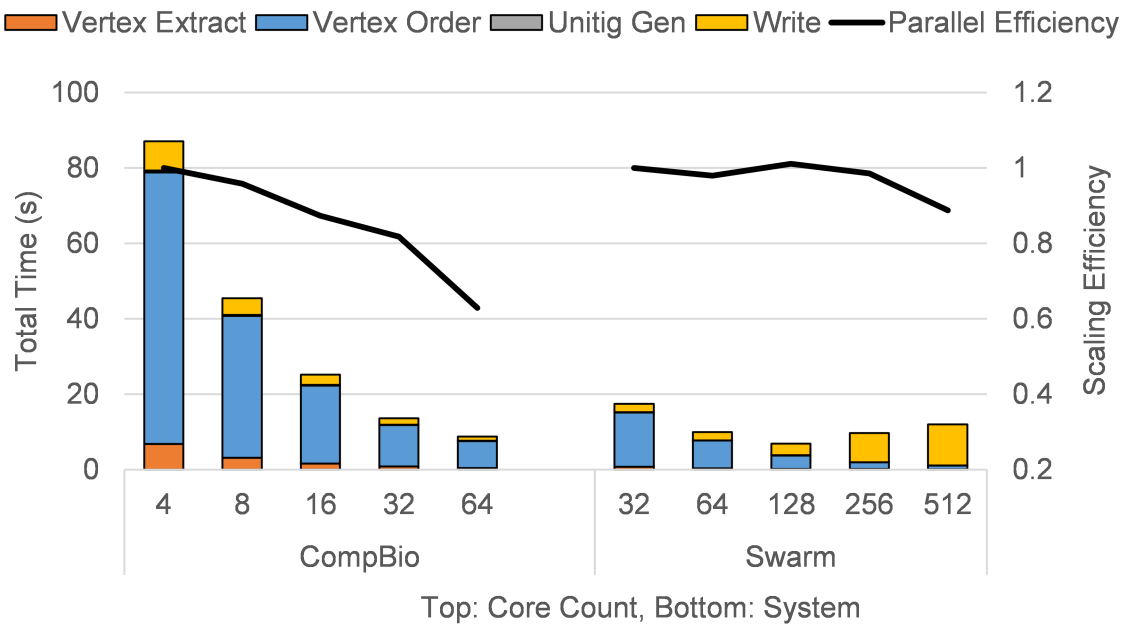



Figure 6.6: Strong scaling behavior of Bruno for compacting de Bruijn graphs for the data set H using 4, 8, 16, 32, and 64 CompBio cores and 2, 4, 8, 16, and 32 Swarm nodes with 16 allocated cores each. The value of $k$ was set to 31, while $f = 4$.

shared memory environments. On CompBio , dominant component is the *vertex ordering* step, while *vertex extraction* and file writing each requiring some non-trivial amount of time. The *vertex ordering* step has parallel efficiency that decreased with increasing $p$, down to $0.62$ at $64$ cores when compared to $4$ cores. We note that at $64$ cores on both CompBio and Swarm , the *vertex ordering* step required approximately the same amount of time, $7.16$ and $7.35$ seconds, suggesting that network communication may not be a significant contributor, and that memory access latency may be a more important factor based on observations from Chapter 3.

Scaling efficiencies on Swarm remained closed to $1$ up to $256$ cores on $16$ nodes, and dropped to $0.89$ for the $512$-core experiments. This again suggest that the compaction process is compute or memory access bound, and network impact is relatively small. The file system performance for Swarm showed strong contention with more than $128$-cores, even with the use of MPI-IO. Subsequent experiments involving Swarm will therefore report

explicitly write times to the GPFS file system.

## 6.4 Comparisons to Existing Unitig Generation Tools

In this section we compare Bruno library's performance for de Bruijn graph construction and chain compaction to HipMer [9] and BCALM2 [59], two state-of-the-art software tools for distributed and shared memory environments, respectively. Only frequency-based error removal is included, as they are the only error removal capabilities provided by HipMer and BCALM2. All Bruno experiments include the times for reading the input, constructing and compacting the graph, generating unitigs, and writing the unitigs to disk, in order to mimic the behavior of HipMer and BCALM2. For both sets of experiments, the reported construction time used the naïve construction algorithm.

### 6.4.1 Distributed Memory Comparison

Table 6.8: Running times in seconds, scaling, and relative speed-ups of our chain compaction algorithm (CC) and the corresponding phases of HipMer (HM) for data sets B (human chromosome 14) and D (human whole genome) on Edison . Scaling values are calculated as the ratio between the times for experiments using the minimal number of cores for a data set, and the number of cores used by the current run. Relative speed-ups between CC and HM are calculated as the ratio between HipMer's times and those of our implementation using the same number of cores.

| | | Time (s) | | | | | | Scaling | | | | | | Speedup HM/CC | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Construct | | Compact | | Total | | Construct | | Compact | | Total | | | | |
| | Cores | CC | HM | CC | HM | CC | HM | CC | HM | CC | HM | CC | HM | Constr. | Comp. | Total |
| B | 96 | 28.1 | 83.0 | 8.3 | 21.4 | 36.4 | 104.4 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 3.0 | 2.6 | 2.9 |
| | 192 | 14.5 | 42.7 | 4.3 | 11.6 | 18.8 | 54.3 | 1.9 | 1.9 | 1.9 | 1.8 | 1.9 | 1.9 | 2.9 | 2.7 | 2.9 |
| | 384 | 7.2 | 21.7 | 2.2 | 5.4 | 9.4 | 27.1 | 3.9 | 3.8 | 3.7 | 4.0 | 3.9 | 3.8 | 3.0 | 2.4 | 2.9 |
| | 768 | 3.8 | 13.9 | 1.4 | 3.5 | 5.2 | 17.4 | 7.3 | 6.0 | 6.0 | 6.1 | 7.0 | 6.0 | 3.6 | 2.6 | 3.3 |
| | 1536 | 2.3 | 15.5 | 0.8 | 2.1 | 3.1 | 17.7 | 12.4 | 5.3 | 10.3 | 10.0 | 11.9 | 5.9 | 6.9 | 2.7 | 5.8 |
| D | 960 | 178.1 | 463.7 | 44.3 | 107.6 | 222.4 | 571.3 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 2.6 | 2.4 | 2.6 |
| | 1920 | 87.7 | 217.9 | 23.1 | 62.5 | 110.8 | 280.4 | 2.0 | 2.1 | 1.9 | 1.7 | 2.0 | 2.0 | 2.5 | 2.7 | 2.5 |
| | 3840 | 44.4 | 126.1 | 12.6 | 35.8 | 57.0 | 161.9 | 4.0 | 3.7 | 3.5 | 3.0 | 3.9 | 3.5 | 2.8 | 2.8 | 2.8 |
| | 7680 | 23.8 | 88.6 | 7.3 | 26.9 | 31.1 | 115.5 | 7.5 | 5.2 | 6.1 | 4.0 | 7.2 | 4.9 | 3.7 | 3.7 | 3.7 |

HipMer [63, 9] is a distributed memory parallel assembler that is currently state-of-the-art in computational performance. Its chain traversal approach is similar to the sparse ruling set strategy for list ranking. Chain compaction in HipMer, the "meraculous" step, proceeds by each processor selecting a random seed $k$-mer and extending it. When two processors try to extend the same chain, one processor yields ownership of the chain. This approach can suffer from frequent communication of small messages and high synchronization costs. We compare the performance and scalability of our algorithm, referred to as **CC** (for **C**hain **C**ompaction) to the equivalent step, "meraculous", in HipMer (**HM**) version 0.9.4.1, using data sets B and D. HipMer was compiled using Berkeley UPC (bupc-narrow) 2.22.3 with default compilation flags. Parameters for HipMer were left as default, and shared memory was used for communication between its graph construction (ufx) and the "meraculous" steps in order to avoid file I/O. HipMer's execution times are extracted from "ufx" and "meraculous" log files. We adopted HipMer's default $k$ and $k$-mer frequency threshold values for our algorithm: $k = 51$ and $f = 4$. The compressed chains were written to disk to mirror HipMer's behavior. For data set B, nodes ranging from 4 to 64 are used, while for data set D, nodes ranging from 40 to 320 are used. With 24-cores per Edison node, this corresponds to tests on processor cores ranging from 96 to 1536 for data set B and from 960 to 7680 for data set D.

To ensure that the performance evaluation are comparable, we verified that each tool operated on approximately the same size graph and produced approximately the same number of unitigs. For data set **D**, $k = 51$ and $f = 4$, HipMer produced $2.829$ billion graph vertices and $22.29$ million unitigs for data set **D**, while Bruno's graph construction and error filtering processes produced $2.815$ billion vertices and $22.20$ million unitigs, representing differences of $0.49\%$ and $0.4\%$.

Table 6.8 shows the running times and scaling behavior of our algorithm and HipMer for both data sets B and D. Our algorithm completed construction and compaction for the human chromosome $14$ read set (B) in $2.3$ and $0.8$ seconds on $1536$ cores, with speedups

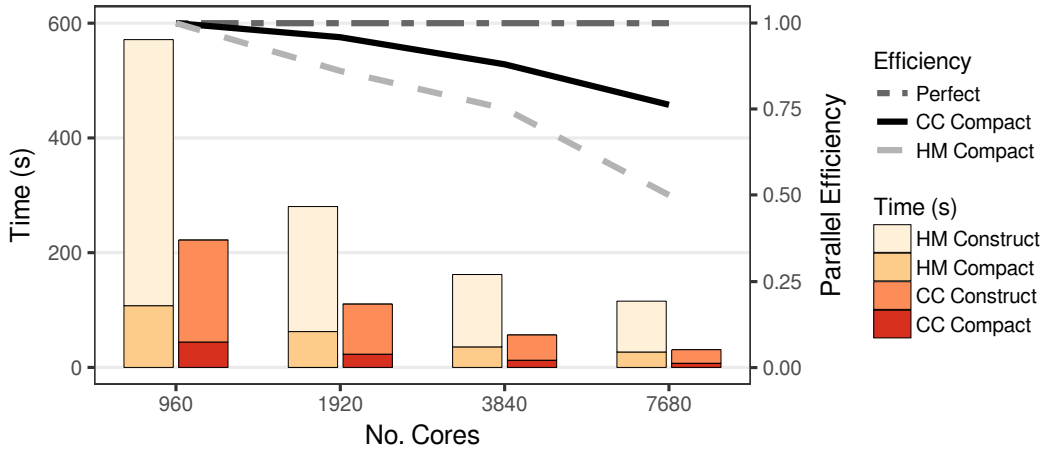



Figure 6.7: Execution time and parallel efficiency for HipMer and our algorithm on $960$ to $7680$ cores, using Edison and data set D. The bars represent the execution times for graph construction and chain compaction, while the lines show parallel efficiency for chain compaction only.

of $12.4\times$ and $10.3\times$ relative to the $96$-core runs, and overall speedup of $11.9\times$. Our implementation therefore achieved a parallel efficiency of $0.64$ for chain compaction and $0.74$ overall when scaling from $96$ cores to $1536$ cores. While HipMer achieved similar speedup and parallel efficiency for unitig generation, its speed up of $5.9\times$ translates to overall parallel efficiency of only $0.37$. Our algorithm was consistently between $2.4\times$ and $2.7\times$ faster than HipMer for chain compaction and is at least $2.9\times$ faster end-to-end for all core counts. For the human read set (D), our implementation completed graph construction and chain compaction in $23.8$ and $7.3$ seconds on $7680$ cores, representing speed ups of $7.5\times$ and $6.1\times$ respectively when compared to the running times on $960$ cores, and a total speedup of $7.2\times$ with $8\times$ more cores. Parallel efficiencies of our algorithm were $0.76$ for chain compaction and $0.9$ overall at $7680$ cores. In contrast, HipMer achieved speed ups of $4.0\times$ for chain compaction and $4.9\times$ overall, with a parallel efficiency of $0.5$ and $0.61$ respectively. Our implementation is up to $3.7\times$ faster than HipMer at $7680$ cores, and at least $2.4\times$ faster for graph construction, chain compaction, and overall times for all core counts. Figure 6.7 further illustrates that our algorithm consistently out-performs HipMer in absolute time and parallel efficiency as core count increases.

### 6.4.2 Shared Memory Comparison

BCALM 2 [59] is the state-of-the-art shared memory chain compaction tool that uses lexicographically minimal substrings within each $k$-mer to partition the $k$-mers. Partitions are stored on disk, and are compacted independently and in parallel, after which the chains from different partitions are merged. We evaluated our algorithm and BCALM 2 (**BC**) on CompBio with data sets B and C. For our algorithm, we treated CompBio as a distributed memory system and assigned one MPI process per core. We adopted the parameters used in [59], $k = 55$ and $f = 3$. The compressed chains were written to disk to mirror BCALM 2's behavior. In contrast to our algorithm, BCALM 2 does not explicitly construct a de Bruijn graph. We therefore measure and compare the total execution times for both BCALM 2 and our algorithm.

For data set C, both the Bruno graph construction process and BCALM 2 produced $2.65$ billion graph vertices or input $k$-mers. Bruno's compaction algorithm generated $17.84$ million unitigs and identified $9.49$ million branch $k$-mers. BCALM 2, on the other hand, produced $23.69$ million unitigs. We note that based on the algorithm description in the BCALM 2 publication, a branch vertex may be output as a unitig by itself or merged with a 5' or 3' neighbor unitig if the connection is unambiguous. Assuming uniform distribution for in- and out-degrees of branch vertices, the probability of encountering branch vertices with $1$ in-edge and more than $1$ out-edges, or $1$ out-edge and more than $1$ in-edges, is $12/21 = 0.57$. This implies that of the $9.49$ million branch vertices reported by Bruno, $9.49 * 0.57 = 5.4$ million $k$-mers may be merged by BCALM2 into unitigs, while the remaining $4.09$ million branch vertices may be reported as standard-alone unitigs. Including this estimate into Bruno's unitig count we arrive at $22.93$ million unitigs, compared to BCALM 2's $23.69$ million unitigs. Given the simplistic statistical assumption, the difference in adjusted unitig counts plausibly indicate that BCALM 2 and Bruno produce equivalent results.

Table 6.9 shows the running times and scaling behavior of our algorithm and BCALM 2,

Table 6.9: Total running time, scalability, and speed up results for (CC) and BCALM 2 (BC) for data set B and C on CompBio . $CC_c$ represents time for chain compaction only. For data set C, only the 64-core results are presented due to long running time. Scalability values (CC, BC) are calculated by comparing to execution time of BCALM 2 and our algorithm on 1 core. Speed-ups of our algorithm relative to BCALM 2 (BC/CC) are calculated for the same number of cores.

| | | Time (s) | | | Scaling | | |
|---|---|---|---|---|---|---|---|
| | Cores | $CC_c$ | CC | BC | CC | BC | BC/CC |
| B: Chr 14 | 1 | 677.2 | 2029.3 | 811.0 | 1.0 | 1.0 | 0.4 |
| | 2 | 376.6 | 1152.1 | 459.1 | 1.8 | 1.8 | 0.4 |
| | 4 | 180.5 | 566.7 | 279.3 | 3.6 | 2.9 | 0.5 |
| | 8 | 91.1 | 289.5 | 192.5 | 7.0 | 4.2 | 0.7 |
| | 16 | 48.7 | 156.2 | 144.8 | 13.0 | 5.6 | 0.9 |
| | 32 | 25.6 | 81.4 | 122.9 | 24.9 | 6.6 | 1.5 |
| | 64 | 15.3 | 47.7 | 116.8 | 42.6 | 6.9 | 2.4 |
| C | 64 | 1352.1 | 2290.3 | 3213.8 | – | – | 1.4 |

as well as the speedup achieved by our algorithm relative to BCALM 2. Our algorithm constructed the graph and compacted chains for data set B and C in $47.7$ seconds and $38.2$ minutes using $64$ cores, approximately $2.4\times$ and $1.4\times$ faster than the corresponding execution times of BCALM 2. Our algorithm scaled significantly better than BCALM 2, achieving a $42.6\times$ speed-up on $64$ cores versus $1$ core, representing a parallel efficiency of $0.67$. In contrast, BCALM 2 was only able to achieve a $6.9\times$ speed up at the same core count, or a parallel efficiency of $0.11$. While BCALM 2 sports better performance when using low core counts, our algorithm achieves similar performance using $16$ cores and outperforms BCALM 2 at higher core counts.

### 6.5 Graph Structure-based Error Detection and Removal

In contrast to frequency-based error detection and removal that is applied during construction, or post-construction but operates on vertices individually, graph structure-based error detection such as bubble and dead-end removal require regional context in the graph. Our algorithm depends on chain compaction to simplify access such contextual information

about the graph.

In addition, application-specific criteria for detecting and removal bubbles and dead-ends imply that the performance of the iterative, multi-pass error removal process necessarily depend on application heuristics. For this evaluation, we devised a simple set of heuristics for identifying and removing structural errors to illustrate the capability of the Bruno library. These heuristics should not be considered definitive nor canonical.

For this evaluation, dead-ends are required to have a maximum length of $k$, while bubbles must have a minimum length of $k$. Additionally, for dead-ends, the point of connection to the rest of the de Bruijn graph must have frequency equal or less than $f$+1, where $f$ is the frequency threshold applied during construction. Bubbles are modified if at last one chain has an edge frequency of $f$+1 between the chain terminal vertex and its branch neighbor. We perform a simple removal of the qualified chain. The graph is then re-compacted. The detection/removal/re-compaction process repeat until no qualifying dead-ends or bubbles can be found in the graph.

### 6.5.1 Performance Characterization

With the constraint that erroneous $k$-mer and structure removal maintains the integrity of chains, i.e. chains are not split, the re-compaction algorithm (Algorithm 5.4) is designed to compute the new chains in a hierarchical manner with $O(|\mathbb{C}|)$ vertices involved in the iterative communication rounds and the total $O(|V_c|)$ vertices accessed and updated only outside the iterative process. We note that if the constraint does not hold, then the standard compaction algorithm must be used.

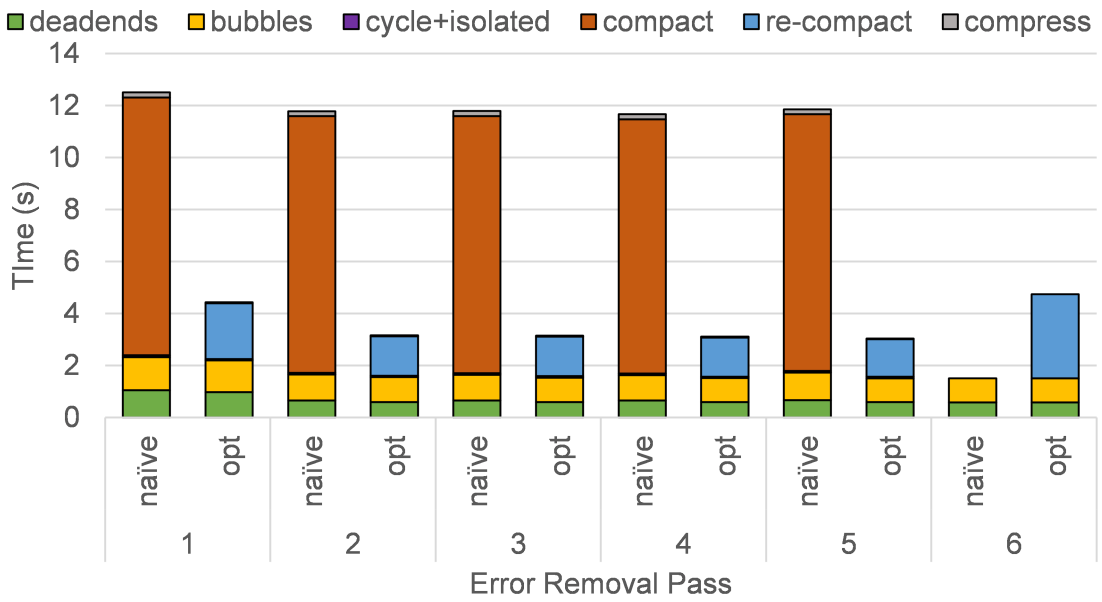
We begin our evaluation by profiling the error removal process for data set I, using $k = 31$ and $f = 2$ running on $16$ Swarm nodes each with $16$ cores allocated to the application, for a total of $256$ cores. For this configuration, the data set required $5$ error removal passes to reach the state where no additional dead-ends and bubbles are found. Including the initial compaction process, the *vertex ordering* operation was invoked $6$ times.

Inspecting the number of vertices ordered by the *vertex ordering* operation and the number of communication iterations it required revealed that the naïve re-compaction process required significant time. Both quantities remain approximately the same, at $10$ for communication iterations, and approximately $3.3$ million vertices per MPI process order through each of the $5$ error removal passes. The number of communication rounds actually increased by $1$ when compared to the initial chain compaction run, as expected due to chains merging and consequently larger maximum chain length.

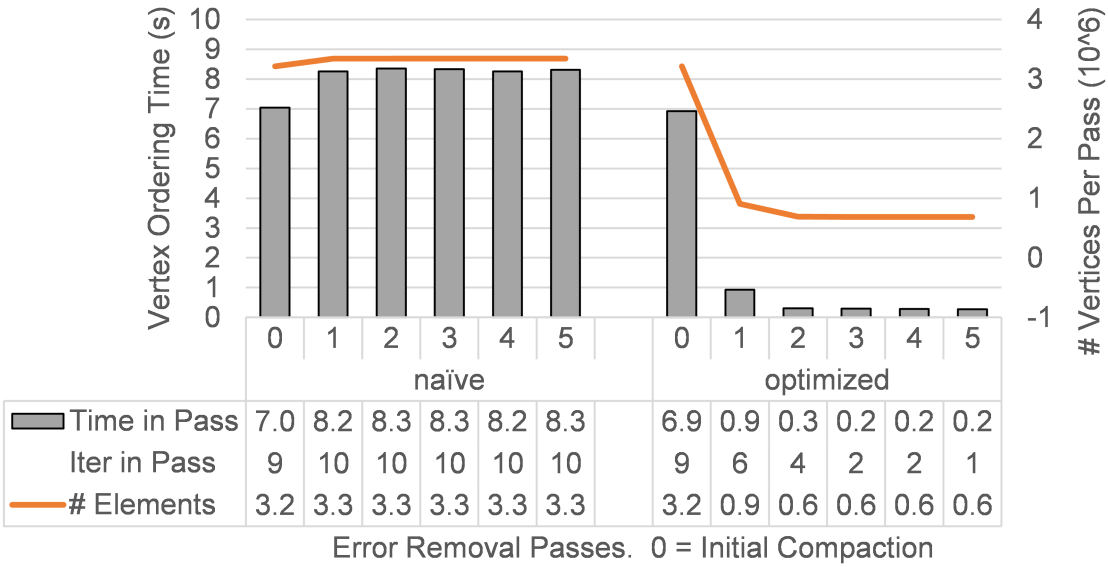
The optimized re-compaction algorithm, on the other hand, showed a decrease of vertex elements from $3.21$ million per MPI process to $0.69$ million, a reduction of $4.6\times$. We note that the element count stabilized after pass 2 as there are significantly fewer modified graph vertices. The required number of communication rounds reduced quickly from $9$ to $2$ within $3$ error correction passes. Consequently, pass $5$ of vertex ordering required only $0.27$ seconds, compared to $8.31$ seconds for the full compaction process, a reduction of over $30\times$.

This reduction in the size of the input to the *vertex ordering* operation and its decreased number of communication iterations per invocation were predicted in Section 5.4.4, and demonstrated experimentally, as shown in Figure 6.8. Note that in Figure 6.8a, pass $6$ involves counting dead-ends and bubbles, whose counts are $0$, and therefore compaction is not invoked in the naïve case. For the optimized case, the re-compaction time actually involves updating all the interval chain vertices in the initial compaction, and therefore *vertex ordering* operation is not involved. Hence in Figure 6.8b pass $6$ is not represented. Instead, Figure 6.8b includes the *vertex ordering* invocation during the initial compaction as a baseline for both the naïve and optimize re-compaction cases.

Table 6.10 shows the time breakdown of each step and the total time for the multi-pass error removal process, aggregated over all passes for $k = 31$ and $f = 2$. We observed that the optimized re-compaction algorithm showed benefits for dead-end, bubble, and cycle detection and removal steps, in addition to re-compaction, due to reduced input size. The




(a) Per pass breakdown of times




(b) Per pass element count and communication rounds

Figure 6.8: Detailed characterization of error removal times for data set I, processed on 256 Swarm cores using $k = 31$ and $f = 2$. 6.8a shows the decomposition of time consumed by each error removal pass, while 6.8b shows the time and sources of improvements for the *vertex ordering* operation during the optimized re-compaction step vs the naïve compaction operation. "Iter in pass" counts the number of communication rounds during an error removal pass, while "element count" is the number of chain vertices used as input for the *vertex ordering* operation.

total speed up reached $2.83\times$ for the optimized version, a significant contributor of which was the re-compaction time, which was reduced from $49.4$ seconds down to $11.4$ seconds with $4.3$

Table 6.10: Comparison between the naïve and optimized re-compaction algorithms. The times are summarized over 5 passes of error removal. Each step in the error removal process is reported, as is the total amount of times. *Dead-ends*, *bubbles*, *cycles+isolated* refer to steps related to the removal of each of these types of erroneous structures. *Re-compaction* includes the time taken to re-compute the compacted chains using the optimized re-compaction algorithm, while during *compress* the compressed chain representation is generated. The total time and parallel efficiency are also reported. Data set I was used, with $k$=31 and $f$=2. The experiments was performed on Swarm using $256$ cores.

| | dead-ends | bubbles | cycles+ isolated | re-compact | compress | **total** | **speed up** |
|---|---|---|---|---|---|---|---|
| naïve | 4.23 | 6.26 | 0.31 | 49.35 | 0.94 | 61.08 | |
| optimized | 3.91 | 5.89 | 0.27 | 11.37 | 0.17 | 21.61 | 2.83 |

For all subsequent experiments involving graph structure-based error removal, the optimized re-compaction algorithm is used.

### 6.5.2 Parameter Evaluation

In our demonstration configuration, the definition of bubbles and chains depend directly on the frequency threshold chosen for the bottom-up frequency-based error filtering (selecting chains with edge frequency below $f$+1), and indirectly on the resulting structure of the graph. In this section we examine the performance impact of the frequency threshold parameter on graph cleaning performance.

*Frequency Filtering Thresholds*

We examined the effects of varying the filtering threshold on the computational performance of bubble and dead-end removal. For this set of experiments, we used $16$ Swarm nodes with $16$ allocated cores each for a total of $256$ cores. The experiments used data set I with parameters $k = 31$ and frequencies $f = \{1, 2, 4\}$.

Table 6.11 shows the total times over all error removal passes for each step, as well as the time and the number of error removal passes used for the entire error removal process. Figure 6.9 illustrates the time break-down by step for each of the removal pass.

Table 6.11: Times for the steps in multi-pass graph structure-based error removal for different frequency threshold values. *Dead-ends*, *bubbles*, *cycles+isolated* refer to steps related to the removal of each of these types of erroneous structures. *Re-compaction* includes the time taken to re-compute the compacted chains using the optimized re-compaction algorithm, while during *compress* the compressed chain representation is generated. The total time and number of error removal passes used are also reported.

| f | dead-ends | bubbles | cycles+ isolated | re-compaction | compress | **total** | **passes** |
|---|---|---|---|---|---|---|---|
| 1 | 15.57 | 23.31 | 1.75 | 50.97 | 0.55 | 92.15 | 4 |
| 2 | 3.91 | 5.89 | 0.27 | 11.37 | 0.17 | 21.61 | 5 |
| 4 | 0.87 | 1.27 | 0.08 | 2.90 | 0.02 | 5.15 | 3 |

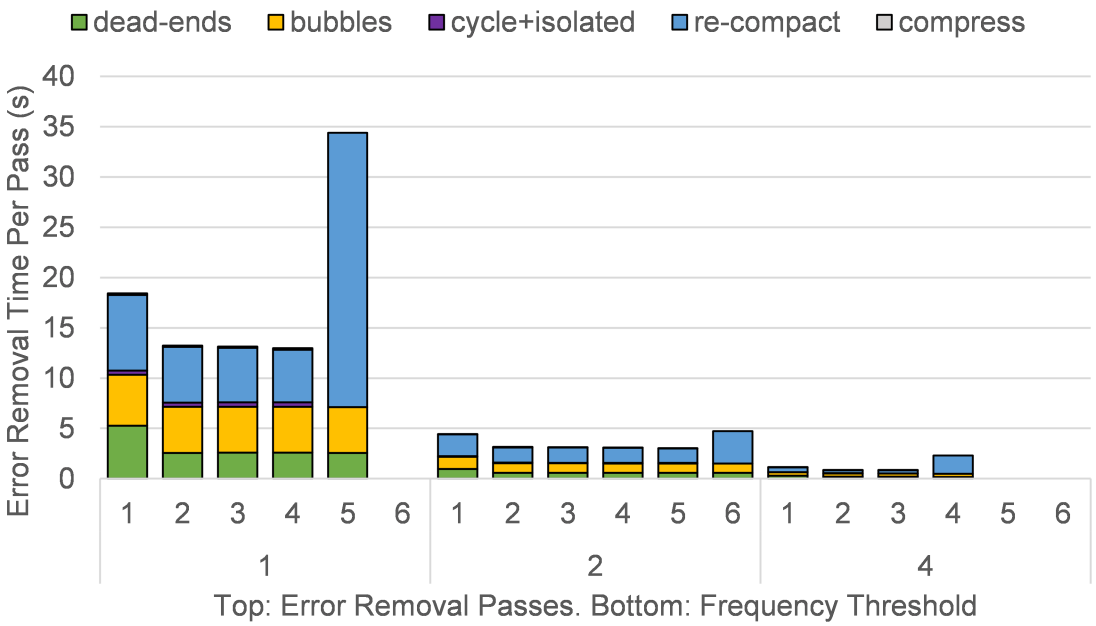



Figure 6.9: Error removal times in seconds on 256 Swarm cores with frequency thresholds of 1, 2, and 4. Each bar indicates the time taken by a error removal pass, decomposed into the times for each step within the pass. Data set H was used with $k = 31$. Missing bars indicate that error removal completed in the previous pass.

The number of error removal passes required depend on the frequency threshold chosen.

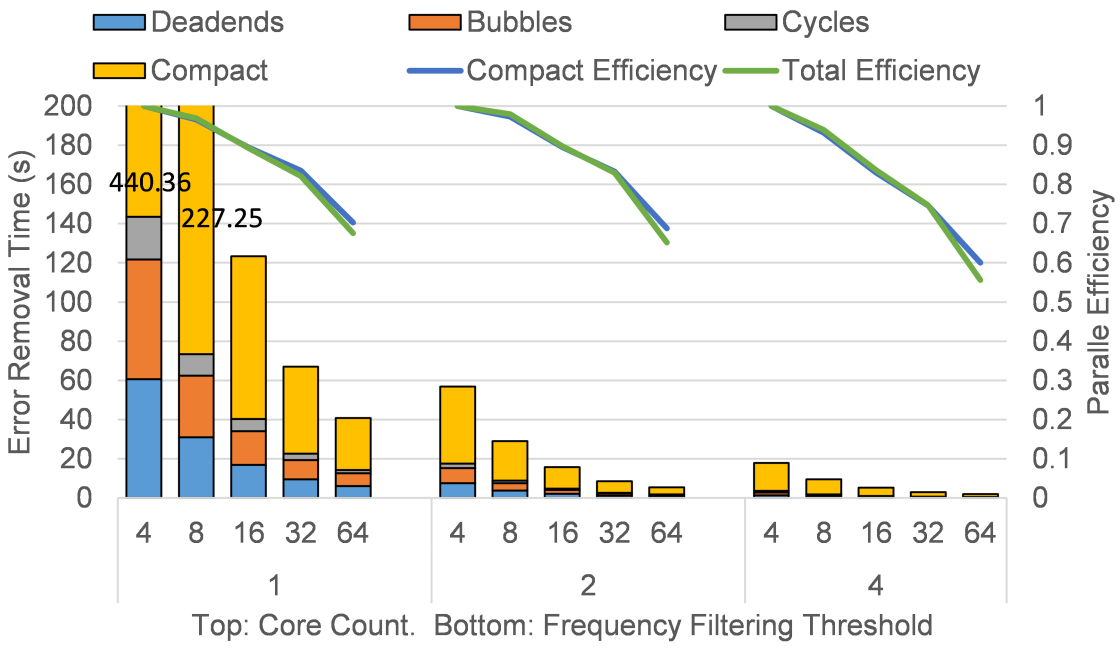
With $f = 2$, 5 passes was needed, while $f = 4$ only 3 passes are required. This variation is due to the fact that bottom-up frequency-based error removal during graph construction already removed a significant number of erroneous $k$-mers that may create the erroneous graph structures. With $f = 1$, edges were not removed through frequency-based filtering, so the bubbles and dead-ends are removed solely based on their structural definition and a rudimentary chain edge frequency filter of 2. The last pass includes a significantly longer re-compaction time. As stated earlier, the initial internal chain nodes are updated to their final state in this invocation. The number of elements updated is therefore significantly higher at approximately the total number of initial chain vertices.

The overall effects of increasing the frequency threshold is dramatic, as the total times as well as the times for each individual steps within the passes showed a 4 to 5-fold decrease for each increase in $f$ tested. Since this parameter directly affects the quality and correctness of the generated unitigs, and is an application specific parameter, we present the results with the goal of estimating the run time contribution of graph structure-based error removal, and not as a guidance for choosing $f$.

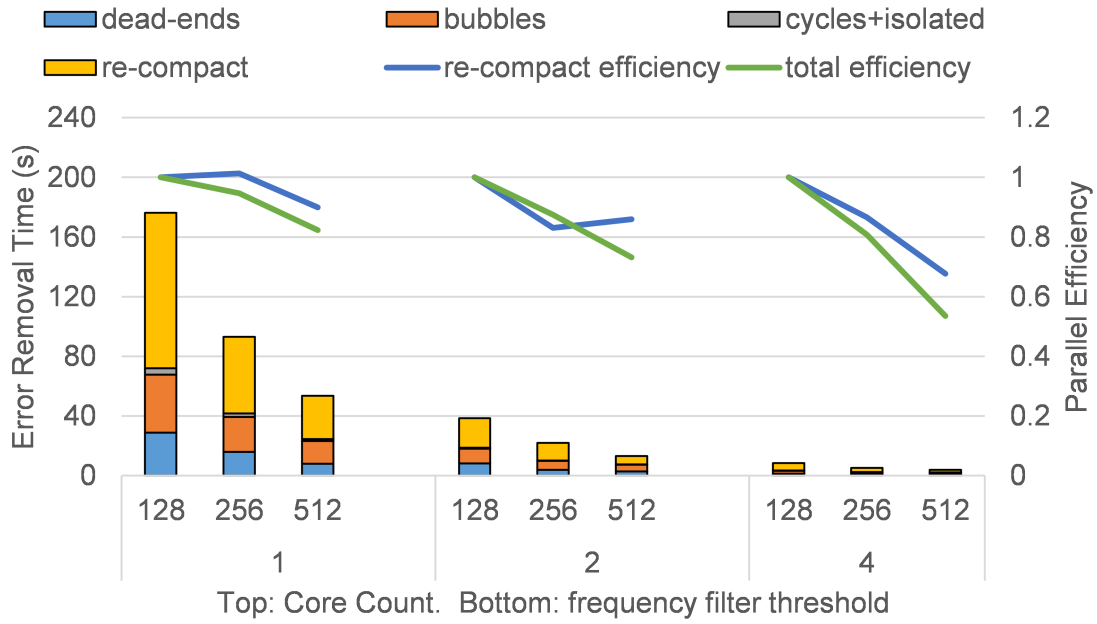
### 6.5.3 Scaling

We evaluate the scalability of the graph structure-based error removal algorithm using data set H on the CompBio system and data set I on the Swarm system. The value of $k$ is set to 31, while the frequency filter is set to $f = 2$. For experiments on the CompBio system, 4, 8, 16, 32, and 64 core are used, while those on the Swarm system, 8, 16, and 32 nodes each with 16 allocated cores were used.

Figure 6.10 and Table 6.12 show the results of the experiments. We were able to complete the error removal process for data set I in 3.9 seconds using 512 distributed memory cores, and 2.1 seconds for data set H using 64 shared memory cores. We note that the error removal process performance scaled sub-linearly as $p$ increased. Overall parallel efficiency decreased relatively quickly at the maximum tested core counts at between 0.54 and 0.56.




(a) Scaling of error removal for data set H on CompBio




(b) Scaling of error removal for data set I on Swarm

Figure 6.10: Scaling behavior of the graph structure-based error removal process on shared (CompBio ) and distributed memory (Swarm ) systems. Data sets H and I were used respectively. Parallel efficiency for the re-compaction step and the error removal process overall are shown as blue and green lines, respectively. The time is broken down into those for different steps. $k = 31$ and $f = \{1, 2, 4\}$.

Table 6.12: Total times for graph structure-based error removal and scaling efficiency for data sets H on CompBio and I on Swarm . The value of $k$ is set to $31$ and frequency is set to $1$, $2$, and $4$

| | | Error Removal Time (s) | | | Parallel Scaling Efficiency | | |
|---|---|---|---|---|---|---|---|
| | f | 1 | 2 | 4 | 1 | 2 | 4 |
| CompBio , H | 4 | 440.36 | 56.80 | 17.90 | 1.00 | 1.00 | 1.00 |
| | 8 | 227.25 | 29.00 | 9.53 | 0.97 | 0.98 | 0.94 |
| | 16 | 123.37 | 15.82 | 5.34 | 0.89 | 0.90 | 0.84 |
| | 32 | 67.05 | 8.56 | 3.00 | 0.82 | 0.83 | 0.75 |
| | 64 | 40.75 | 5.45 | 2.01 | 0.68 | 0.65 | 0.56 |
| Swarm , I | 128 | 176.21 | 38.40 | 8.37 | 1.00 | 1.00 | 1.00 |
| | 256 | 93.06 | 21.95 | 5.18 | 0.95 | 0.87 | 0.81 |
| | 512 | 53.49 | 13.13 | 3.91 | 0.82 | 0.73 | 0.54 |

This is likely due to the random memory access during *vertex ordering* and the relatively high communication activities in that step. In addition, bubble and dead-end detection required parallel sorting, which has logarithmic complexity for the local sorting. Figure 6.10 shows that while *vertex ordering* was the majority contributor to the overall time, dead-end and bubble detection and removal also required substantial fraction of the total time.

Finally, the timing results reiterates that the choice of $f$ is important, and that using a frequency filtering threshold that is larger than $1$ is beneficial for computational performance of the compaction and error removal process. As removal of erroneous $k$-mer is a common pre-processing step, we recommend that $f$ be set to some value larger than $1$ but in accordance with the application requirements.

## 6.6 Unitig Quality After Error Removal

Given a set of reads and value $k$, a de Bruijn graph can be deterministically constructed. Chain compaction is a deterministic process, therefore the compacted chains can be ideally determined based on teh graph structure alone, in the absence of any modifications from frequency-based filtering, such as those described in Section 5.3, or structure-based modifications including bubble and dead-end removal (Section 5.4).

However, in the presence of such graph modifications, the resulting compacted chains or unitigs will change. In this section we evaluated the quality, or more appropriately, infer the correctness of the generated unitigs by comparing them to the corresponding reference genomes.

We use data sets F, G, and H for this evaluation. These data sets are from the GAGE [93] project and include reference genomes. The unitigs are generated using frequency thresholds $f = \{1, 2, 3, 4, 8\}$. Each experiment is run with frequency filtering only, and with frequency filter and bubble and dead-end removal. The experiments were conducted on CompBio using $64$ cores and the unitig files save in multi-FASTA files.

The QUAST [94] assembly quality measurement tool is used to assess the unitigs. In evaluation, we are interested in assessing the correctness of the compaction, as well as the efficacy of the frequency filtering and bubble and dead-end removal in increasing the unitig lengths. For the first task, we are interested in the numbers of mis-assemblies, unaligned unitigs, and whether the maximum aligned lengths match the maximum unitig lengths. For the second task, we look for increases in N50 as well as an increase in the number of unitigs over $500$, $1000$, and $5000$ bps.

Table 6.13 shows the results of running QUAST. Related to the correctness of the generated unitigs, we observed that for all read sets and all frequency and error removal configurations Bruno achieved $0$ or $1$ mis-assembly, and less than $10$ unaligned unitigs for the chromosome 14 data set. For bacterial data sets, there were no unaligned unitigs. The maximum size of the aligned unitigs match the maximum unitig size exactly, indicating that graph structure-based error removal is performing without errors. Coupled with the low misalignment and unaligned counts, this observation suggests that most or all unitigs were computed correctly.

With respect to increasing frequency filtering thresholds, we observed that the distribution of unitig lengths increased with increasing $f$ for *H. sapiens* chromosome 14 and *R. sphaeroides* data sets. For *S. aureus*, increasing the frequency threshold to $8$ actually

decreased the overall unitig lengths. The choice of $f$ therefore needs to be tailored for the data set at hand, based on genome and sequencing coverage.

We observed also that there is a positive effect of structure-based error removal, but the effect are non-uniform. For the human data set, we observed increases in unitigs larger than $1000$ and $5000$, but the N50 value increased only slightly. Similar trend is present for the two bacterial data sets.

Overall, the QUAST evaluation indicate that Bruno's construction, compaction, and error removal processes perform correctly. The choice of frequency threshold is therefore critical, and should be assessed based on application requirements and data characteristics. Generally, structure-based error removal improves the unitig lengths and therefore it is recommended that this step be applied, provided that the criteria for identifying bubbles and dead-ends are defined carefully.

Table 6.13: Evaluation of unitigs generated after frequency-based filtering (*freq*) and topological structure filtering, including bubble, dead-end, and cycle removals (*freq + topo*), for the human chromosome 14, *R. sphaeroides*, and *S. aureus* sequence data sets. QUAST was used with default parameters.

| | | | # unitigs longer than specified base pairs | | | | unitig length | | | | # | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $f$ | $\geq 0$ | $\geq 500$ | $\geq 1000$ | $\geq 5000$ | N50 | Max | Max Aligned | Mis-assem | Unaligned | % Genome |
| *H. sapiens* chr14 | freq | 2 | 9447756 | 122 | 0 | 0 | 553 | 827 | 827 | 0 | 0 | 0.065 |
| | | 3 | 3219547 | 20881 | 2130 | 0 | 695 | 3639 | 3639 | 1 | 6 | 13.855 |
| | | 4 | 2040068 | 35787 | 6552 | 3 | 793 | 5300 | 5300 | 0 | 4 | 26.555 |
| | | 8 | 1177994 | 47022 | 13752 | 25 | 965 | 9064 | 9064 | 0 | 3 | 40.633 |
| | freq+topo | 1 | 43930232 | 1 | 0 | 0 | 679 | 679 | 679 | 0 | 0 | 0.001 |
| | | 2 | 6713989 | 7743 | 252 | 0 | 616 | 2036 | 2036 | 1 | 6 | 4.589 |
| | | 3 | 2607266 | 33793 | 5544 | 2 | 770 | 5300 | 5300 | 1 | 6 | 24.454 |
| | | 4 | 1798014 | 41851 | 9605 | 8 | 861 | 7803 | 7803 | 0 | 5 | 33.119 |
| | | 8 | 1133617 | 47717 | 14756 | 37 | 997 | 9064 | 9064 | 0 | 3 | 42.273 |
| *R. sphaeroides* | freq | 3 | 289150 | 85 | 0 | 0 | 558 | 889 | 889 | 0 | 0 | 1.041 |
| | | 4 | 129964 | 938 | 38 | 0 | 626 | 1532 | 1532 | 0 | 0 | 13.159 |
| | | 8 | 43187 | 2683 | 564 | 0 | 833 | 3062 | 3062 | 1 | 0 | 47.963 |
| | freq+topo | 2 | 625501 | 60 | 0 | 0 | 551 | 889 | 889 | 0 | 0 | 0.738 |
| | | 3 | 189116 | 1026 | 44 | 0 | 629 | 1722 | 1722 | 0 | 0 | 14.443 |
| | | 4 | 94141 | 2126 | 256 | 0 | 722 | 2056 | 2056 | 0 | 0 | 33.44 |
| | | 8 | 39279 | 2808 | 689 | 1 | 881 | 5007 | 5007 | 1 | 0 | 52.73 |
| *S. aureus* | freq | 2 | 255131 | 32 | 0 | 0 | 575 | 861 | 861 | 0 | 0 | 0.643 |
| | | 3 | 66705 | 2143 | 777 | 1 | 1049 | 5557 | 5557 | 0 | 0 | 72.096 |
| | | 4 | 40247 | 1703 | 1012 | 20 | 1848 | 8932 | 8932 | 0 | 0 | 87.236 |
| | | 8 | 45905 | 1605 | 422 | 0 | 926 | 4647 | 4647 | 0 | 0 | 50.285 |
| | freq+topo | 2 | 175793 | 2131 | 833 | 1 | 1097 | 5557 | 5557 | 0 | 0 | 74.669 |
| | | 3 | 51920 | 1396 | 967 | 70 | 2473 | 9754 | 9754 | 0 | 0 | 91.122 |
| | | 4 | 34056 | 1460 | 955 | 48 | 2335 | 12147 | 12147 | 0 | 0 | 89.933 |
| | | 8 | 44370 | 1607 | 425 | 0 | 927 | 4988 | 4988 | 0 | 0 | 50.48 |

## 6.7 Summary

In this chapter we evaluated the performance of the graph construction, compaction, and error removal algorithm implementations in the Bruno library. We showed that Bruno was able to compact human-genome-size de Bruijn graphs in $7.3$ seconds using $7680$ cores on a distributed memory system and in $22.5$ minutes using $64$ shared memory cores. It outperformed two state-of-the-art assembly tools for shared memory and distributed memory environments by $1.4\times$ and $3.7\times$ respectively including construction and compaction times.

Error removal in a graph constructed from an $162$ GB data set completed in $13.1$ and $3.91$ seconds with frequency filter of $2$ and $4$ respectively on $16$ nodes, totaling $512$ cores.

Evaluation of the unitig quality suggest that the error removal process is correct and produces only negligible number of mis-assemblies and unaligned unitigs, while qualitatively improves the unitig lengths.

# CHAPTER 7
# CONCLUSION

As next generation sequencing become more affordable and ubiquitous, the downstream sequence analysis increasingly require higher performance, lower latency, and better scalability to handle the ever increasing volume of data at ever increasing generation rates. Distributed memory systems provide such scalability in both computation resources and memory space for data. Yet a significant number of bioinformatic tools remain bound to single-node systems.

In this research, we sought to facilitate the adoption and increase the utilization of distributed memory environments in bioinformatics algorithms and applications, through the development of high performance, distributed memory parallel algorithms and libraries for $k$-mer indexing and counting, and de Bruijn graph and its operations. K-mer counting and indexing are central to many bioinformatics applications such as *de novo* assembly, genome mapping, and error correction, while de Bruijn graphs are central to *de novo* genome assembly and thus is a critical first step in genomic data analysis.

We developed two distributed memory libraries, Kmerind for $k$-mer indexing, and Bruno for de Bruijn graph operations, towards satisfying our research aims:

1. Create the first reusable, flexible, and extensible distributed memory parallel libraries, with simple API and optimized implementations, for $k$-mer indexing and de Bruijn graphs for bioinformatics applications.

2. Develop efficient data structure and algorithms specifically for distributed memory parallel $k$-mer counting and indexing

3. Develop efficient data structure for distributed memory parallel de Bruijn graphs, and efficient algorithms for graph construction, compaction, traversal, and error removal.

Kmerind, to our knowledge, is the first generic $k$-mer indexing library for distributed-memory environments. We designed Kmerind with a simple set of API with clear sequential semantics to facilitate adoption, and designed and implemented efficient distributed memory parallel data structures and algorithms to support this API.

Bruno, to our knowledge, is the first generic de Bruijn graph library for distributed-memory environments. It abstracts common functionalities useful for genome assembly and other de Bruijn graph based applications, and provides construction, chain compaction, and error removal operations on a de Bruijn graph. The API has been defined at 3 levels, corresponding to operations on vertices and edges, chains, and whole graph. Through composition of these operations, the graph can be traversed, and higher level operations can be constructed.

In both libraries, the algorithm and implementations that support the APIs have been designed for distributed memory from the ground up. In the case of Kmerind, architecture aware optimizations have been applied to further improve the efficiency of the overall algorithms.

Performance evaluations throughout this work have shown that Kmerind and Bruno out-perform the current state-of-the-art tools for $k$-mer counting and de Bruijn graph construction and chain compaction in both distributed and shared memory environments.

### 7.0.1 Broader Applicability

While Kmerind and Bruno target bioinformatic applications, the algorithms and implementations presented in this dissertation are significant to computer science and applicable to a broader set of domains.

First, our model of linear chains in bi-directed de Bruijn graphs can be considered as a generalization of doubly linked lists to allow nodes with reversed orientations. It can alternatively be considered as a specialization of doubly linked lists with undirected nodes and edges, of which linear chains in undirected graphs are an example. Our *vertex*

*ordering* and *re-compaction* algorithms, when frame in such context, can be considered as generalized list ranking or undirected graph compaction algorithms provided 1) the $d$-path formulation that encapsulates $k$-mer representation is adapted to the graph formulation at hand, and 2) an implicit but computable vertex and edge ordering can be defined.

Second, while our *cycle detection* algorithm currently serves as a stopping criterion for the *vertex ordering* algorithm, it is useful in its own right when detecting cycle presence is the primary objective. Cycles may also be labeled using the lexicographically minimal $k$-mers seen during *vertex ordering*, as suggested in Section 4.2.6. Since doubling and traversal occurs on each chain and cycle separately, cycle vertices would be labeled with the minimal $k$-mers within each chain.

Extending this idea further, the *vertex ordering* and *cycle detection* algorithms can be modified to detect and label cycles within a certain size range if we annotate each $d$-path with the lexicographically minimal $k$-mers seen in the 5' and 3' directions. During iteration $t$, the lexicographically minimal $k$-mer in the 5' side of the $d$-path is the minimal $k$-mer amongst the $2^t$ $k$-mers 5' to the vertex being examined, based on the associative property of the minimum operation, and similarly for the 3' lexicographically minimal $k$-mer. For a vertex in a cycle of size $|C| < 2^{t+1}$,the 5' and 3' lexicographically minimal $k$-mers are identical as the $d$-path spans the entire cycle. This property allows each cycle of size $|C|$ to be detected and labeled during iteration $t = \log(|C|) - 1$.

Our *vertex ordering* and *cycle detection* algorithms therefore has broad applicability to graph analysis, as together they allow the detection and identification of cycles in a graph, as well as classification of cycles by size, given a suitable definition of the $d$-path abstraction and an associative operator for identifying minimal vertex id, both of which are expected to be simple and direct.

Our optimized implementations are useful beyond our target $k$-mer counting and indexing and de Bruijn graph use cases. The distributed memory hash tables in Kmerind have been designed with generic templated API (Chapter 2). The key, hash function and

comparison operations are customizable. They have already been applied as generic tuple indices with integer keys. Similarly, the optimize Robin Hood hashing based hash table (Section 3.2.3) is a generic templated implementation that can be used beyond $k$-mer indexing, provided the application can support batch mode hash table operations. The vectorized MurmurHash 3 hash functions (Section 3.4.2) likewise is useful in general. The implementations are designed to work with short byte arrays. In testing, it has been shown to work well with arrays of integers, each representing a key.

Finally, the Kmerind and Bruno libraries as a whole may have direct applications outside of the bioinformatics domain. N-gram modeling in computational linguistics, for example, involves statistical analysis of languages and texts using the occurrences of n-grams, which are defined similarly to $k$-mers, as signatures. A critical difference lies in the alphabet size. While genomic sequences typically uses a small alphabet and therefore can be represented compactly, natural language processing often has to be able to handle significantly larger alphabet size, such as that for East Asian languages. Kmerind therefore may be useful for n-gram modeling, provided the alphabet size does not become a critical limiting factor.

### 7.0.2 Open Problems

The Kmerind and Bruno libraries have been designed for distributed memory $k$-mer counting and indexing, and de Bruijn graph operations including construction, compaction, and error removal. For these tasks, they have been shown to perform and scale well in shared and distributed memory environments. However, there are several areas where future research efforts should be directed.

One immediate area concerns Kmerind and Bruno's integration with existing and novel bioinformatics applications, as well as the continued evolution of its APIs and optimization of their algorithms and implementations. As we encounter new use cases, the libraries may need to be extended to address differences in data characteristics and application require-

ments, for example to identify the presence of rare variants using de Bruijn graphs, and usage in long-read analyses. A related investigation involves the development of language bindings, for example with perl, python, or R.

A second aspect involves efficient memory utilization. While distributed memory computing has allowed Bruno and Kmerind to leverage large compute and memory capacities and to scale to larger data, scaling down to smaller systems is limited first and foremost by the available memory. Memory efficient data structures have been studied extensively for the $k$-mer counting problem in share memory environment, and frequent approaches, as described in Chapter 1, include the use of disks as external memory and probabilistic therefore inexact data structures such as CountMinSketch. The design and integration of associative data structures that are performant, cache friendly, scalable to distributed memory, exact, yet memory efficient can have a significant impact on Bruno and Kmerind's adoption by applications that operate in shared memory environments.

The library APIs for Kmerind and Bruno have been designed for distributed but tightly coupled systems, such as clusters. Meanwhile, the availability and low barrier of entry for cloud based systems has prompted increased adoption of such models of computation in bioinformatics. For similar reasons, the use of hardware accelerators such as GPUs have become increasingly common. The mapping of Kmerind and Bruno APIs and algorithms to such platforms can be useful in improving performance, simplifying usage, unifying application development efforts, and increasing the libraries' adoption. Evaluation of the feasibility in mapping between the computation and communication models would be useful.

## REFERENCES


[1] The Cancer Genome Atlas Research Network, J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. M. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, and J. M. Stuart, "The Cancer Genome Atlas Pan-Cancer analysis project," *Nature Genetics*, vol. 45, no. 10, pp. 1113–1120, 2013.

[2] The 1000 Genomes Project Consortium, "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, 2015.

[3] K.-P. Koepfli, B. Paten, t. G. K. C. o. Scientists, and S. J. OBrien, *The Genome 10k Project: A Way Forward*, review-article, Feb. 2015.

[4] S. Caboche, C. Audebert, and D. Hot, "High-Throughput Sequencing, a VersatileWeapon to Support Genome-Based Diagnosis in Infectious Diseases: Applications to Clinical Bacteriology," *Pathogens*, vol. 3, no. 2, pp. 258–279, 2014.

[5] A. Geskin, E. Legowski, A. Chakka, U. R. Chandran, M. M. Barmada, W. A. LaFramboise, J. Berg, and R. S. Jacobson, "Needs Assessment for Research Use of High-Throughput Sequencing at a Large Academic Medical Center," *PLOS ONE*, vol. 10, no. 6, e0131166, 2015.

[6] P. Muir, S. Li, S. Lou, D. Wang, D. J. Spakowicz, L. Salichos, J. Zhang, G. M. Weinstock, F. Isaacs, J. Rozowsky, and M. Gerstein, "The real cost of sequencing: Scaling computation to keep pace with data generation," *Genome Biology*, vol. 17, p. 53, 2016.

[7] D. R. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de bruijn graphs," *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.

[8] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "Abyss: A parallel assembler for short read sequence data," *Genome Research*, vol. 19, no. 6, pp. 1117–1123, 2009.

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

[10] 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.

[11] D. R. Kelley, M. C. Schatz, and S. L. Salzberg, "Quake: Quality-aware detection and correction of sequencing errors," *Genome Biology*, vol. 11, no. 11, p. 1, 2010.

[12] X. Yang, K. S. Dorman, and S. Aluru, "Reptile: Representative tiling for short read error correction," *Bioinformatics*, vol. 26, no. 20, pp. 2526–2533, 2010.

[13] R. Ounit, S. Wanamaker, T. J. Close, and S. Lonardi, "Clark: Fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers," *BMC Genomics*, vol. 16, p. 236, 2015.

[14] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[15] W. J. Kent, "BLATthe blast-like alignment tool," *Genome Research*, vol. 12, no. 4, pp. 656–664, 2002.

[16] 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.

[17] F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp, "Mrsfast: A cache-oblivious algorithm for short-read mapping," *Nature Methods*, vol. 7, no. 8, pp. 576–577, 2010.

[18] Y. Li, J. M. Patel, and A. Terrell, "Wham: A high-throughput sequence alignment method," *ACM Transactions on Database Systems*, vol. 37, no. 4, p. 28, 2012.

[19] G. Myers, "Efficient local alignment discovery amongst noisy long reads," in *International Workshop on Algorithms in Bioinformatics*, Springer, 2014, pp. 52–67.

[20] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov, and E. S. Lander, "Arachne: A whole-genome shotgun assembler," *Genome Research*, vol. 12, no. 1, pp. 177–189, 2002.

[21] 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. 1, 2011.

[22] N. Philippe, M. Salson, T. Lecroq, M. Leonard, T. Commes, and E. Rivals, "Querying large read collections in main memory: A versatile data structure," *BMC Bioinformatics*, vol. 12, no. 1, p. 1, 2011.

[23] N. Välimäki and E. Rivals, "Scalable and versatile k-mer indexing for high-throughput sequencing data," in *International Symposium on Bioinformatics Research and Applications*, 2013, pp. 237–248.

[24] Y. Li *et al.*, "MSPKmerCounter: A fast and memory efficient approach for k-mer counting," *ArXiv preprint arXiv:1505.06550*, 2015.

[25] A.-A. Mamun, S. Pal, and S. Rajasekaran, "Kcmbt: A k-mer Counter based on Multiple Burst Trees," *Bioinformatics*, vol. 32, no. 18, pp. 2783–2790, Sep. 2016.

[26] 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.

[27] G. Rizk, D. Lavenier, and R. Chikhi, "Dsk: K-mer counting with very low memory usage," *Bioinformatics*, vol. 29, no. 5, pp. 652–653, 2013.

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

[29] 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, e101271, 2014.

[30] R. S. Roy, D. Bhattacharya, and A. Schliep, "Turtle: Identifying frequent k-mers with cache-efficient algorithms," *Bioinformatics*, vol. 30, no. 14, pp. 1950–1957, 2014.

[31] 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.

[32] M. Erbert, S. Rechner, and M. Mller-Hannemann, "Gerbil: A fast and memory-efficientk-mer counter with GPU-support," *Algorithms for molecular biology: AMB*, vol. 12, p. 9, 2017.

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

[34] *Kmernator: An MPI Toolkit for large-scale genomic analysis*, Accessible at `https://github.com/JGI-Bioinformatics/Kmernator`, accessed on Apr 4, 2016.

[35] R. Chikhi and P. Medvedev, "Informed and automated k-mer size selection for genome assembly," *Bioinformatics*, vol. 30, no. 1, pp. 31–37, Jan. 2014.

[36] P. Melsted and B. V. Halldrsson, "Kmerstream: Streaming algorithms for k-mer abundance estimation," *Bioinformatics*, btu713, Oct. 2014.

[37] G. Cormode and S Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[38] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[39] 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*, vol. PP, no. 99, pp. 1–1, 2017.

[40] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

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

[42] P. Flick and S. Aluru, "Parallel distributed memory construction of suffix and longest common prefix arrays," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015, p. 16.

[43] Y. Liu, B. Schmidt, and D. L. Maskell, "Parallelized short read assembly of large genomes using de Bruijn graphs," *BMC Bioinformatics*, vol. 12, no. 1, p. 354, 2011.

[44] P. A. Pevzner, M. Y. Borodovsky, and A. A. Mironov, "Linguistics of nucleotide sequences II: Stationary words in genetic texts and the zonal structure of DNA," *Journal of Biomolecular Structure and Dynamics*, vol. 6, no. 5, pp. 1027–1038, 1989.

[45] P. A. Pevzner, H. Tang, and M. S. Waterman, "An Eulerian path approach to DNA fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[46] J.-i. Sohn and J.-W. Nam, "The present and future of de novo whole-genome assembly," *Briefings in Bioinformatics*, bbw096, 2016.

[47] 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.

[48] R. Luo, B. Liu, Y. Xie, Z. Li, W. Huang, J. Yuan, G. He, Y. Chen, Q. Pan, Y. Liu, J. Tang, G. Wu, H. Zhang, Y. Shi, Y. Liu, C. Yu, B. Wang, Y. Lu, C. Han, D. W. Cheung, S.-M. Yiu, S. Peng, Z. Xiaoqian, G. Liu, X. Liao, Y. Li, H. Yang, J. Wang, T.-W. Lam, and J. Wang, "Soapdenovo2: An empirically improved memory-efficient short-read de novo assembler," *GigaScience*, vol. 1, no. 1, p. 18, 2012.

[49] Y. Peng, H. C. Leung, S.-M. Yiu, and F. Y. Chin, "IDBA–a practical iterative de Bruijn graph de novo assembler," in *Annual International Conference on Research in Computational Molecular Biology*, Springer, 2010, pp. 426–440.

[50] J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, and D. S. Rokhsar, "Meraculous: De novo genome assembly with short paired-end reads," *PLOS ONE*, vol. 6, no. 8, e23501, 2011.

[51] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski, *et al.*, "SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing," *Journal of Computational Biology*, vol. 19, no. 5, pp. 455–477, 2012.

[52] D. Antipov, A. Korobeynikov, J. S. McLean, and P. A. Pevzner, "HybridSPAdes: An algorithm for hybrid assembly of short and long reads," *Bioinformatics*, btv688, 2015.

[53] S. Gnerre, I. MacCallum, D. Przybylski, F. J. Ribeiro, J. N. Burton, B. J. Walker, T. Sharpe, G. Hall, T. P. Shea, S. Sykes, A. M. Berlin, D. Aird, M. Costello, R. Daza, L. Williams, R. Nicol, A. Gnirke, C. Nusbaum, E. S. Lander, and D. B. Jaffe, "High-quality draft assemblies of mammalian genomes from massively parallel sequence data," *Proceedings of the National Academy of Sciences*, vol. 108, no. 4, pp. 1513–1518, 2011.

[54] A. V. Zimin, G. Marçais, D. Puiu, M. Roberts, S. L. Salzberg, and J. A. Yorke, "The MaSuRCA genome assembler," *Bioinformatics*, vol. 29, no. 21, pp. 2669–2677, 2013.

[55] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome Research*, vol. 22, no. 3, pp. 549–556, 2012.

[56] T. C. Conway and A. J. Bromage, "Succinct data structures for assembling large genomes," *Bioinformatics*, vol. 27, no. 4, pp. 479–486, 2011.

[57] C. Ye, Z. S. Ma, C. H. Cannon, M. Pop, and W. Y. Douglas, "Exploiting sparseness in de novo genome assembly," *BMC Bioinformatics*, vol. 13, no. 6, p. 1, 2012.

[58] K. Salikhov, G. Sacomoto, and G. Kucherov, "Using cascading Bloom filters to improve the memory usage for de Brujin graphs," *Algorithms for Molecular Biology*, vol. 9, no. 1, p. 1, 2014.

[59] 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.

[60] S. Boisvert, F. Laviolette, and J. Corbeil, "Ray: Simultaneous assembly of reads from a mix of high-throughput sequencing technologies," *Journal of Computational Biology*, vol. 17, no. 11, pp. 1519–1533, 2010.

[61] B. G. Jackson, M. Regennitter, X. Yang, P. S. Schnable, and S. Aluru, "Parallel de novo assembly of large genomes from high-throughput short reads," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, Apr. 2010, pp. 1–10.

[62] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "SWAP-Assembler: Scalable and efficient genome assembly towards thousands of cores," *BMC Bioinformatics*, vol. 15, no. 9, S2, 2014.

[63] 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*, ser. SC '14, New Orleans, Louisana: IEEE Press, 2014, pp. 437–448.

[64] T. El-Ghazawi and L. Smith, "Upc: Unified parallel C," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, 2006, p. 27.

[65] L. Zeng, J. Cheng, J. Meng, B. Wang, and S. Feng, "Improved Parallel Processing of Massive De Bruijn Graph for Genome Assembly," in *Web Technologies and Applications: 15th Asia-Pacific Web Conference, APWeb 2013, Sydney, Australia, April 4-6, 2013. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 96–107.

[66] V. K. Kundeti, S. Rajasekaran, H. Dinh, M. Vaughn, and V. Thapar, "Efficient parallel and out of core algorithms for constructing large bi-directed de Bruijn graphs," *BMC Bioinformatics*, vol. 11, no. 1, p. 560, 2010.

[67] J. C. Wyllie, "The Complexity of Parallel Computations," Ithaca, NY, USA, Tech. Rep., 1979.

[68] M. Reid-Miller and G. E. Blelloch, "List Ranking and List Scan on the CRAY C-90," Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep., 1994.

[69] R. Cole and U. Vishkin, "Faster optimal parallel prefix sums and list ranking," *Information and Computation*, vol. 81, no. 3, pp. 334–352, Jun. 1989.

[70] R. J. Anderson and G. L. Miller, "Deterministic parallel list ranking," *Algorithmica*, vol. 6, no. 1-6, pp. 859–868, Jun. 1991.

[71] F. Dehne and S. W. Song, "Randomized parallel list ranking for distributed memory multiprocessors," *International Journal of Parallel Programming*, vol. 25, no. 1, pp. 1–16, 1997.

[72] J. F. Sibeyn, F. Guillaume, and T. Seidel, "Practical Parallel List Ranking," *Journal of Parallel and Distributed Computing*, vol. 56, no. 2, pp. 156–180, 1999.

[73] L. Ilie, B. Haider, M. Molnar, and R. Solis-Oba, "Sage: String-overlap assembly of genomes," *BMC Bioinformatics*, vol. 15, no. 1, p. 302, Sep. 2014.

[74] G. M. Kamath, I. Shomorony, F. Xia, T. A. Courtade, and D. N. Tse, "Hinge: Long-read assembly achieves optimal repeat resolution," *Genome Research*, vol. 27, no. 5, pp. 747–756, 2017.

[75] A. Döring, D. Weese, T. Rausch, and K. Reinert, "SeqAn An efficient, generic C++ library for sequence analysis," *BMC Bioinformatics*, vol. 9, p. 11, 2008.

[76] *Message Passing Interface (MPI)*, Accessible at `http://mpi-forum.org`, accessed April 4, 2016.

[77] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A comparison of sorting algorithms for the connection machine cm-2," in *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, ACM, 1991, pp. 3–16.

[78] P. Flick, *Mxx: C++11 Message Passing*, Accessible at `https://github.com/patflick/mxx`, accessed Mar 7, 2017.

[79] R. J. Fisher and H. G. Dietz, "Compiling for simd within a register," in *International Workshop on Languages and Compilers for Parallel Computing*, Springer, 1998, pp. 290–305.

[80] *Sparsehash: C++ associative containers*, Accessible at `https://github.com/sparsehash/sparsehash`, accessed Nov 1, 2016.

[81] V. Shulaev, D. J. Sargent, R. N. Crowhurst, T. C. Mockler, O. Folkerts, A. L. Delcher, P. Jaiswal, K. Mockaitis, A. Liston, S. P. Mane, P. Burns, T. M. Davis, J. P. Slovin, N. Bassil, R. P. Hellens, C. Evans, T. Harkins, C. Kodira, B. Desany, O. R. Crasta, R. V. Jensen, A. C. Allan, T. P. Michael, J. C. Setubal, J.-M. Celton, D. J. G. Rees,

K. P. Williams, S. H. Holt, J. J. R. Rojas, M. Chatterjee, B. Liu, H. Silva, L. Meisel, A. Adato, S. A. Filichkin, M. Troggio, R. Viola, T.-L. Ashman, H. Wang, P. Dharmawardhana, J. Elser, R. Raja, H. D. Priest, D. W. B. Jr, S. E. Fox, S. A. Givan, L. J. Wilhelm, S. Naithani, A. Christoffels, D. Y. Salama, J. Carter, E. L. Girona, A. Zdepski, W. Wang, R. A. Kerstetter, W. Schwab, S. S. Korban, J. Davik, A. Monfort, B. Denoyes-Rothan, P. Arus, R. Mittler, B. Flinn, A. Aharoni, J. L. Bennetzen, S. L. Salzberg, A. W. Dickerman, R. Velasco, M. Borodovsky, R. E. Veilleux, and K. M. Folta, "The genome of woodland strawberry (Fragaria vesca)," *Nature Genetics*, vol. 43, no. 2, pp. 109–116, Feb. 2011.

[82] B. Nystedt, N. R. Street, A. Wetterbom, A. Zuccolo, Y.-C. Lin, D. G. Scofield, F. Vezzi, N. Delhomme, S. Giacomello, A. Alexeyenko, *et al.*, "The norway spruce genome sequence and conifer genome evolution," *Nature*, vol. 497, no. 7451, pp. 579–584, 2013.

[83] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd. The MIT Press, 2009.

[84] R. Pagh and F. F. Rodler, "Cuckoo Hashing," in *Algorithms ESA 2001*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Aug. 2001, pp. 121–133.

[85] M. Herlihy, N. Shavit, and M. Tzafrir, "Hopscotch Hashing," in *Distributed Computing*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Sep. 2008, pp. 350–364.

[86] P. Celis, "Robin Hood Hashing," Ph.D. Thesis, University of Waterloo, Waterloo, ON, Canada, 1986.

[87] *Robin Hood hashing: Backward shift deletion | Code Capsule*, Accessible at `http://codecapsule.com/2013/11/17/robin-hood-hashing-backward-shift-deletion`, accessed Feb 21, 2017.

[88] M. Ankerl, *Very Fast HashMap in C++: Benchmark Results (Part 3)*, Accessible at `https://martin.ankerl.com/2016/09/21/very-fast-hashmap-in-c-part-3`, accessed Aug 23, 2017.

[89] A. Fog, *Software optimization resources. C++ and assembly. Windows, Linux, BSD, Mac OS X*, Accessible at `http://www.agner.org/optimize`, accessed on Oct 18, 2017.

[90] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13, Genoa, Italy: ACM, 2013, pp. 683–692, ISBN: 978-1-4503-1597-5.

[91] P. Flajolet, ric Fusy, O. Gandouet, and et al., "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in *In AOFA 07: Proceedings of the 2007 International Conference on Analysis of Algorithms*, 2007.

[92] P. Medvedev, K. Georgiou, G. Myers, and M. Brudno, "Computability of Models for Sequence Assembly," in *Algorithms in Bioinformatics*, 00139, Springer, Berlin, Heidelberg, Sep. 2007, pp. 289–301.

[93] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, G. Marais, M. Pop, and J. A. Yorke, "Gage: A critical evaluation of genome assemblies and assembly algorithms," *Genome Research*, vol. 22, no. 3, pp. 557–567, 2012.

[94] A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler, "Quast: Quality assessment tool for genome assemblies," *Bioinformatics*, vol. 29, no. 8, pp. 1072–1075, Apr. 2013, 00893.

# VITA

Tony Pan received an Sc. B. degree in Biophysics from Brown University in 1998 and an M.S. degree in Computer Science from Rensselaer Polytechnic Institute in 2000. He began his PhD studies part-time in the School of Computational Science and Engineering at Georgia Institute of Technology in 2010, and became full-time in 2013.

Prior to his PhD studies, Mr. Pan held research staff positions at General Electric, The Ohio State University, and Emory University. He developed biomedical imaging applications ranging from CT visualization and visual analytic 3D digital microscopy, to large scale microscopy and pathology image segmentation and classification in HPC and cloud environments. Mr. Pan was also involved in national software infrastructure efforts including the National Cancer Institute's Cancer Bioinformatics Grid project, serving as software architect for radiology and pathology imaging applications and grid middleware systems.

Mr. Pan's research interests include high performance computing and parallel algorithms, computational data analysis, bioinformatics, biomedical informatics, imaging informatics, and distributed information systems. He has authored or co-authored over 32 journal articles and 23 conference publications, as well as 6 book chapters and 4 patents.