# Parallelizing Irregular Applications for Distributed Memory Scalability: Case Studies from Genomics

*Marquita Ellis*
*Katherine A. Yelick, Ed.*
*James Demmel, Ed.*
*Aydin Buluç, Ed.*
*Daniel Rokhsar, Ed.*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-133
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-133.html

June 3, 2020

# Parallelizing Irregular Applications for Distributed Memory Scalability: Case Studies from Genomics

by

Marquita May Ellis

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Katherine Yelick, Chair
Professor James Demmel
Professor Daniel Rokhsar
Adjunct Assistant Professor Aydın Buluç

Spring 2020

Parallelizing Irregular Applications for Distributed Memory Scalability: Case Studies from Genomics

Abstract

Parallelizing Irregular Applications for Distributed Memory Scalability: Case Studies from Genomics

by

Marquita May Ellis

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Katherine Yelick, Chair

Generalizable approaches, models, and frameworks for irregular application scalability is an old yet open area in parallel and distributed computing research. Irregular applications are particularly hard to parallelize and distribute because, by definition, the pattern of computation is dependent upon the input data. With the proliferation of data-driven and data-intensive applications from the realm of Big Data, and the increasing demand for and availability of large-scale computing resources through HPC-Cloud convergence, the importance of generalized approaches to achieving irregular application scalability is only growing.

Rather than offering another software language or framework, this dissertation argues we first need to understand application scalability, especially irregular application scalability, and more closely examine patterns of computation, data sharing, and dependencies. As it stands, predominant performance models and tools from parallel and distributed computing focus on applications that are divided into distinct communication and computation phases, and ignore issues related to memory utilization. While time-tested and valuable, these models are not always sufficient for understanding full application scalability, particularly, the scalability of data-intensive irregular applications. We present application case studies from genomics, highlighting the interdependencies of communication, computation, and memory capacities and performance.

The genomics applications we will examine offer a particularly useful and practical vantage point for this analysis, as they are data-intensive irregular application targets for both HPC and cloud computing. Further, they present an extreme for both domains. For HPC, they are less akin to traditional, well-studied and well-supported scientific simulations and more akin to text and document analysis applications. For cloud computing, they are an extreme in that they require frequent random global access to memory and data, stressing interconnection network latency and bandwidth and co-scheduled processors for tightly orchestrated

computation.

We show how common patterns of irregular all-to-all computation can be managed efficiently, comparing bulk-synchronous approaches built on collective communication and asynchronous approaches based on one-sided communication. For the former, our work is based on the popular Message Passing Interface (MPI) and makes heavy use of globally collective communication operations that exchange data across processors in a single step or, to save memory use, in a set of irregular steps. For the latter, we build on the UPC++ programming framework, which provides lightweight RPC mechanisms, to transfer both data and computational work between processors. We present performance results across multiple platforms including several modern HPC systems and, at least in one case, a cloud computing platform.

With these application case studies, we seek not only to contribute to discussions around parallel algorithm and data structure design, programming systems, and performance modeling within the parallel computing community, but also to contribute to broader work in genomics through software development and analysis. Thus, we develop and present the first distributed memory scalable software for analyzing data sets from the latest generation of sequencing technologies, known as *long read* data sets. Specifically, we present scalable solutions to the problem of *many-to-many long read overlap and alignment*, the computational bottleneck to long read assembly, error correction, and direct analysis. Through cross-architectural empirical analysis, we identify the key components to efficient scalability, and highlight the priorities for any future optimization with analytical models.

To the two people who have continuously supported and inspired me from the first, in ways and depths that words cannot express: Carmen and Lorenna, my mom and sister.

# Contents

# List of Figures

# List of Tables

# Acknowledgments

I cannot thank Kathy Yelick, my advisor, enough; I have learned so much from her expertise and example, yet I believe I could spend years learning more. I also thank my committee, Jim Demmel, Aydın Buluç, and Dan Rokhsar, for their valuable feedback and investment in this work.

I can only begin to list the people I would like to thank for their support in so many varied, technical and non-technical, ways: first, Maurice Herlihy, who introduced me to parallel computing and pointed me down the road to a research career – I may have missed that sign post if not for him, and his fascinating research and teaching on concurrency and synchronization; my other professors at Brown University, especially John Savage and Andy van Dam, for their advice and encouragement toward graduate school; my academic siblings at Berkeley who preceded me, especially, Evangelos Georganas and Penporn Koanantakool, and those who succeed me, especially, Giulia Guidi, Yang You, and Benjamin Brock; my peers, Emmeline Kao, Ph.D., and Kayla Wolf, Ph.D, for their camaraderie and inspiring examples in science and engineering; Paul Riggins, Ph.D., for his valuable perspectives on scientific communications; Dr. Ira Young, Audrey Sillers, Shirley Salanio, among so many other faculty and staff who work, seemingly tirelessly, to support graduate students and recruit diverse thinkers, enriching the department, the university, and the world; Joseph Brown, for so patiently explaining for-loops to me in my first computer science class; my students and mentees, including my little sisters in WICSE and AWE, for reminding me why continuing to push forward is so important, even beyond the technical contributions.

This work was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration as part of the ExaBiome project at Lawrence Berkeley National Laboratory and by the National Science Foundation as part of the SPX program under Award number 1823034 at UC Berkeley. This research used resources of the National Energy Research Scientiffic Computing Center (NERSC) under contract No. DE-AC02-05CH11231, and the Oak Ridge Leadership Computing Facility under DE-AC05-00OR22725. all supported by the Office of Science of the U.S. Department of Energy. The information presented here does not necessarily reflect the position or the policy of the Government and no offcial endorsement should be inferred.

# Chapter 1

# Introduction

Several important trends in parallel and distributed computing, both long standing and more recent, provide the framework for this dissertation work. These include the proliferation of Big Data applications; the increasing heterogeneity in computer architectures; the phenomena known as HPC-Cloud convergence; and the challenge of mapping irregular applications onto parallel machines. A more specific question is the debate over whether, when, and how bulk-synchronous parallelism offers a superior model to asynchronous parallelism and vice versa. We consider these challenges and questions concretely with application case studies of important bioinformatics applications. The broader impact of this work is in supporting bioinformaticians working toward new energy, food, environmental sustainability, and medical discoveries, among so many other areas of worldwide interest. We shall briefly describe each of the computing trends mentioned, beginning with HPC-cloud convergence, and then outline specific application case studies.

At its inception, cloud computing looked very different from High Performance Computing (HPC). Its purpose was broadening computing accessibility supported by clusters of computers with commodity hardware and networks. HPC on the other hand has historically supported compute-intensive scientific discovery, on supercomputers with special purpose hardware and highest-speed interconnects. For a long time, the two subfields developed in different directions, working with different design goals and constraints, both physical and financial. In recent years, however, particularly with the rise of Big Data, we have been observing the phenomena known as HPC-Cloud convergence. Cloud providers, faced with massive and growing demand for both small and large scale computing resources, have been adapting techniques and machinery developed for HPC. The "Tech Giants" of today including Google, Microsoft, and Amazon have all made notable strides in this direction. Demand for large scale resources is growing with the availability of data sets from all sectors of science and society, along with increasing knowledge and motivation for analyzing it. This increase in data and data analysis tools has also impacted HPC. In the past, HPC primarily supported scientific simulation. Currently, however, the ability to analyze massive data sets is just as important, from analyzing Cosmic Microwave Background data to Large Hadron Collider data to Event Horizon Telescope data. Motivating applications span many

other domains as well, from material science, to genomics, to cross-domain document search and analysis. We expect the HPC-Cloud convergence trend to continue, from the common incentive to support Big Data analysis and the need to adapt to basic constraints of large scale computing such as power consumption, heat dispersion, and so on.

Complicating the architectural design space in both HPC and cloud domains, is the rise of extreme on-board heterogeneity, with network design and interface. Computer architectures have evolved from single-CPU and single memory hierarchy to multi-CPU, multi-core, and many-core designs, with uniform and non-uniform memory access, NIC-on-chip, and a variety of accelerators, from FPGAs to GPUs. With such a large design space, a natural question from computer architecture is, **what is the right balance of computational power, memory rates and capacities, and network performance to support a broad or targeted class of applications?** While this dissertation will not dive deeply into architectural design, we will consider this high-level question for specific irregular applications.

Irregular applications are becoming increasingly important to both cloud and HPC. Hence, an old problem in parallel and distributed computing resurfaces, **what are generalizable approaches, models, and programming paradigms for maximizing irregular application parallelism and scalability?** Irregular applications are particularly challenging to parallelize and distribute because, by definition, the pattern of computation is determined by the input data and is not known until runtime. Irregularity may manifest in the data structures (e.g. unbalanced trees, irregular graphs, unstructured grids), the control flow (e.g. data-dependent branching), and the data access and communication patterns. Irregular application parallelization must therefore be adaptive at runtime (dynamic) and/or rely on domain-specific expertise. The drawbacks of dynamic parallelization and optimization are well-known. In the extreme, dynamic methods solve the problem before they can solve it efficiently. For example, to load balance the computation, communication, and storage for analyzing relationships across input data, one might compute the structure of the associated graph and partition it accordingly; however, then the analyses might be complete. More effective dynamic approaches, that analyze some smaller aspect of the data or runtime behavior, still often yield sub-linear speedups due to their runtime overhead, except under rare and usually architecture-specific conditions. Parallelization approaches that employ domain-specific expertise, on the other hand, have been highly successful in a number of cases. However, they entail two simple inherent drawbacks. One is that they require a domain expert. The other is that they are domain-specific (usually lacking in generalizability), in some cases, even across varying constraints on the same problem. It comes as no surprise that a considerable portion of Big Data applications fall into the category of irregular applications, given their data-driven nature.

Data-intensive applications from bioinformatics offer a valuable vantage point from which to study the intersection of these trends and the related questions. Bioinformatics applications are unlike standard physical or scientific applications, relying on integer and string operations, rather than typical floating point operations. In this regard, they are similar to text and document analyses. However, unlike general document analysis, the impact or

success of biological analyses depends on exact character-to-character level accuracy in the end results. Thus, bioinformatics applications represent an extreme for irregular application analysis. Furthermore, bioinformaticians seek support from both the cloud and HPC facilities to run their analyses, due to various constraints on access and availability. This dissertation therefore works with application case studies from bioinformatics to shed light on these broader questions in computing as well as to support bioinformatics research.

Before moving to the outline and contributions of the dissertation, we briefly visit the last above-mentioned open question. There is a long-standing debate over two competing models of parallel computation, bulk-synchronous parallelism[61] (BSP) and asynchronous parallelism. Asynchronous parallelism offers an arguably more natural way to conceptualize and therefore to program parallel computation graphs. On the other hand, BSP offers a simpler framework for correctness and performance analysis, and is sufficiently expressive for all kinds of computation even if it is a less "natural" for programming certain applications. The debate is particularly relevant for irregular applications. Asynchrony may support a more adaptive style of communication and computation, and therefore be the more natural choice for a given irregular application. However, because successful bulk-synchronous programming requires balancing communication and computation in well-defined steps, any strategies for "regularizing" an irregular application that prove successful under a bulk-synchronous model, are likely useful across paradigms. In any case, such strategies may provide a better starting point for parallelization and optimization. The general question is, **for what kinds of computations or under what circumstances does bulk-synchronous parallelism provide a superior model to asynchronous parallelism, and visa versa?**

In order to contribute to this discussion in computing from an application-grounded standpoint, and to have an even more immediate, positive impact beyond computing, we consider these questions through software development and performance analysis for bioinformatics, more specifically, for genomics. First, we consider two particular but representative types of data-driven graph construction applications for genomics, *de Bruijn* and *overlap* graph construction in Chapter 3 - each will be defined in (Chapters 2-3). We consider a bulk-synchronous approach to these irregular applications, and likewise, the extent to which necessary "regularization" techniques are successful. We find that certain approaches to managing memory go hand-in-hand with communication and computation optimization. For other optimizations in the same case study, we observe trade-offs between communication scalability and computational or memory costs.

In Chapter 4, we present a bulk-synchronous approach to *overlap* graph traversal and refinement. Together with the *overlap* graph construction presented in Chapter 3, we analyze its performance, across several HPC architectures as well as a cloud offering, which offer various balancing points between computation and communication resources. We find a very tight coupling between communication and computation efficiency, that determines overall performance and scalability outcomes across architectures. As a significant contribution of this dissertation, the *overlap* graph construction, traversal, and refinement were implemented by this author in the DiBELLA [13] software pipeline, the first distributed memory scalable software for *long read analysis and alignment* - a problem defined in Chapter 2.

DiBELLA expands our related work on BELLA [23], a well-formalized methodology for accurate long read overlap detection and alignment, that uses global information from across the input. The DiBELLA work focuses on the challenges of parallelizing this application in distributed memory. In addition, the DiBELLA software supports a range of alternative accuracy constraints and models, via user parameters and modular implementation.

In Chapter 5, we revisit *overlap* graph refinement with an asynchronous approach for balancing memory, communication, and computation. We compare the performance of our asynchronous implementation presented in Chapter 5 to our bulk-synchronous implementation from Chapter 4. The analysis sheds light on the different circumstances under which either approach, a bulk-synchronous or an asynchronous approach, to parallelism leads to a more optimal design for this application.

Chapters 3-5 together ultimately present a case for memory, communication, and computation to be considered together in distributed memory performance models for data-intensive parallel applications. As expounded further in the conclusions and future work in Chapter 7, such models would greatly benefit the work of application developers and hardware architects seeking to support irregular and data-intensive applications across domains. Since each of the main chapters 3-5 present algorithms, pertinent implementation details, and analytical and empirical analysis for each case study, related work is presented more fully in Chapter 6.

Finally, the contributions of this dissertation, including algorithms, software, and analytical and empirical performance analysis, are as follows:

1. An in-depth analysis of techniques and optimizations (with their limitations and trade-offs) for enabling bulk-synchronous scalability of two irregular applications from genomics, *short read de Bruijn* and *long read overlap* graph construction (Chapter 3),

2. A first distributed memory scalable, bulk-synchronous approach and implementation to *long read overlap* graph construction, based on successful approaches to extreme-scale *short read de Bruijn* graph construction[18] (Chapter 3),

3. A first distributed memory scalable, bulk-synchronous algorithm for *long read overlap* graph traversal and refinement, with implementation in MPI 2.0 (Chapter 4),

4. A performance analysis of bulk-synchronous *long read overlap* graph construction, traversal, and refinement across several HPC architectures and one cloud offering, highlighting the computation-communication balance and scalability bottlenecks for this problem (Chapter 4),

5. A distributed memory scalable, alternative (asynchronous) algorithm for *long read overlap* graph refinement, with implementation in UPC++ (Chapter 5),

6. An empirical performance comparison of our bulk-synchronous and asynchronous approaches to *long read overlap* graph refinement (Chapter 5),

7. The software pipeline for *long read overlap* graph construction, traversal, and refinement, named DiBELLA (Chapter 3-4),

8. A foundation for future and alternative pipeline optimizations and implementations; a foundation composed of software, performance models, and empirical cross-architecture and cross-workload performance results (Chapter 3-7).

# Chapter 2

# Preliminaries

This chapter establishes terminology and definitions used in the following chapters.

## 2.1 Reads, Errors, and $k$-mers

Genome sequencing technologies translate DNA into sequences of the characters A, C, T, and G, corresponding to the four nucleotide bases, adenine, cytosine, guanine, and thymine. Sometimes, sequencing technologies also emit the character N when a base is detected but its type cannot be determined with confidence. These sequences, strings over the alphabet {A, C, T, G, N}, are referred to simply as *reads*. The literature further distinguishes reads from next generation or shotgun sequencing and from third generation or Single Molecule Real Time [12] sequencing as *short reads* and *long reads*, respectively. The names come from their relative typical lengths. Short reads are typically 100 to 250 characters whereas long reads range from $1,000$ to $100,000$ characters.

Due to limitations in the technology, the confidence with which a sequencer makes a *base call* is not 100%. Sequencer confidence scores for each base are included in FASTQ files, a standard type of input file for our genome analysis tools. An erroneous base call might be, for example, emitting an A in a position where the genome actually has some nucleotide base other than adenine. The confidence score for this call may be low, but it does not tell us which other base might be there. Moreover, errors can also be insertions and deletions of characters. The frequency at which a sequencer emits errors is referred to as the *error rate*. State-of-the-art short read sequencers have very low error rates, from 0.001% [57] to 2% [62], but the distribution of errors is usually unknown. Long read sequencers have much higher error rates, within 5 - 35% historically. However, certain long read sequencers also guarantee uniform distributions of errors [12] - a helpful guarantee for computational models [7][49][23].

A common strategy for coping with or eliminating errors in both types of reads is to parse them into substrings called *k-mers*. *K-mers* are substrings computed by moving a sliding window of length $k$ over the reads, usually one character at a time. Figure 2.1 illustrates computing 6-mers from an example read. Once *k-mers* are computed, certain strategies

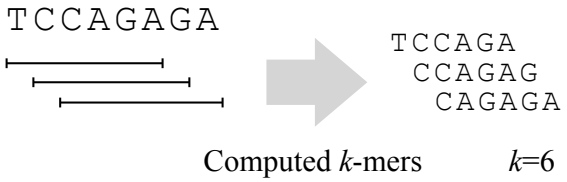


Figure 2.1: Illustrates the computation of the three $k$-mers (6-mers) on the right from the underlined regions of the string on the left.


Figure 2.2: The 6-mers shared between two strings on the left are used to seed the alignment of the same two strings on the right.

attempt to eliminate error-containing *k-mers* based on the frequency of distinct *k-mers*, the confidence scores, inter *k-mer* relationships, and other means. The specific strategies employed depend on the application. In general, the frequencies of distinct *k-mers*, i.e. the *k-mer* histogram for a data set, are often useful in and of itself in bioinformatics. Hence, a wide variety of tools have been created specifically to count *k-mers* and output *k-mer* histograms (see Chapter 6 regarding related work). In the types of genome analyses presented in this dissertation, *k-mer* counting will be employed, but as a feature or component to solving a larger analysis problem (see Chapter 3)

## 2.2 Alignment

The first distributed-memory-scalable *many-to-many pairwise aligner*, diBELLA[13], is a contribution of this dissertation. Chapters 4-5 describe algorithms for bulk synchronous and asynchronous approaches to many-to-many pairwise alignment, and analyze the performance characteristics thereof across architectures. This section provides the necessary definitions and descriptions for understanding the many-to-many pairwise alignment problem and for distinguishing it from other alignment problems.

### Pairwise Alignment

A formal definition of *pairwise alignment* [11][63] for pairs of strings over the alphabet $\Lambda = \{A, C, T, G\}$*[1] is as follows. Let $\Lambda'$ be the alphabet extending $\Lambda$ with the gap character '$-$', that is, $\Lambda' = \Lambda \cup \{-\}$. A pairwise alignment of a string $s \in \Lambda$ of length $m$, and a string $t \in \Lambda$ of length $n$, is the pair of strings $(s', t') \in \Lambda'$ if and only if:

1. $|s'| = |t'| = l$, where $max(m, n) \leq l \leq m + n$,

[1]For concreteness, $\Lambda$ is defined here with respect to DNA. However, the pairwise alignment definition is easily extended for other applications by substituting the alphabet for e.g. RNA, $\Lambda_{RNA} = \{A, U, G, C\}$, or the 24 characters representing amino acids in protein analyses.

2. $s$ is the string obtained by removing all '$-$' from $s'$, and $t$ is the string obtained by removing all '$-$' from $t'$,

3. Letting $s'_i$ indicate the character at position $i$ in string $s'$, and likewise letting, $t'_i$ indicate the character at position $i$ in $t'_i$, such that $1 \leq i \leq l$, there is no value of $i$ for which $s'_i = t'_i = \text{'}-\text{'}$.

In general, there are exponentially many possible valid alignments of two strings, exponential in the length of the strings. However, one is generally interested in only high quality alignments as determined by a *scoring scheme*. A scoring scheme assigns numeric penalties to the following *edit operations*:

- $s'_i = -$, called "insertion"
- $t'_i = -$, called "deletion"
- $s'_i \neq - \wedge\ t'_i \neq -$, called "substitution"

Scoring schemes may also numerically reward matches. The sum of the rewards and penalties for a particular alignment is its score. An alignment is *optimal* if its score is the maximum achievable for the given pair of strings - or the minimal achievable for the dual minimization problem. Finding an optimal alignment is attainable via a dynamic programming algorithm such as Needleman-Wunsch [50] or Smith-Waterman [59], and is an $O(m \cdot n)$ computation. In place of full dynamic programming for pairwise alignment, one can also search only for solutions with a limited number of mismatches (banded Smith-Waterman) and terminate early when the alignment score drops significantly ($x$-drop) [69]. Such approximations usually attain lower average-case complexity bounds by relaxing or modifying guarantees about the resulting alignments.

## Seed-and-Extend Pairwise Alignment with *k-mer* Seeds

Alternatives to exact dynamic programming approaches for pairwise alignment include *seed-and-extend* approaches. These treat a given substring, a "seed", as a partial alignment between the two strings. Figure 2.2 illustrates a simple alignment between two strings seeded with matching *k-mers*. The alignment computation "extends" the alignment by searching forward in each string for an alignment that includes the seed. An alignment that is extended in the opposite direction or both directions may be computed by reversing the strings and the seed and repeating the procedure. The alignment in Figure 2.2 is extended in both directions. The extension in this example simply determines what is done with the leading and trailing two characters around the seed in the respective strings. Even for these short example strings, this seed-and-extend alignment is significantly less costly than computing a full Smith-Waterman or Needleman-Wunsch alignment. The intuition behind the seed-and-extend paradigm is that any read pair that aligns well is likely have such seeds in common. Chapter 6 discusses seeding strategies besides exact-matching *k-mers*. Chapters 4-5 use exact-matching *k-mers* as seeds for pairwise alignment.

## Pairwise versus Multiple Sequence Alignment

*Multiple sequence alignment*[2] is a generalization of pairwise alignment for finding a consensus across a set of two or more sequences (strings). An optimal multiple sequence alignment minimizes or maximizes the alignment score across (all) strings in the set. A series of pairwise alignments do not necessarily produce an optimal multiple-sequence-alignment. Due to the high error rates of long reads and the priority for high accuracy in overlap detection, we employ pairwise alignment on potentially overlapping reads, even when we have detected a set of more than two reads that all potentially overlap. We will expand on this point in later chapters.

## Many-to-Many Pairwise Alignment

One of the key problems addressed in this dissertation is *many-to-many pairwise alignment*. To describe the many-to-many alignment problem, we extend pairwise alignment to sets. Given sets of strings $S$ and $T$, where $S$ and $T$ may be the same, we are interested in all high-scoring pairwise alignments of strings $(s,t) \in \{S \times T\} \mid s \neq t$, given some scoring scheme. The many-to-many alignment problem additionally differs from the straightforward problem of computing all pairwise alignments in $\{S \times T\}$ in that we are only interested in *high-scoring* alignments. In other words, there is an opportunity to reduce the number of pairwise alignment computations across the set if we can detect and filter pairs of strings that are unlikely to have a sufficiently high-scoring alignment. Given that such similarity detection depends on runtime analysis of the input, however, this presents the challenge of balancing the subsequent pairwise alignment computations in distributed memory parallelization.

## One-to-Many or Many-to-One versus Many-to-Many Alignment

In order to clarify the challenges distributed many-to-many alignment, we distinguish it from a category of alignment algorithms and software that we dub "one-to-many" or "many-to-one" aligners. These include software tools that align many reads to a single reference; we can think of a reference as long string representing the whole genome, the result of previous assembly. For example, one may seek an alignment against a human reference genome. Many algorithms and software tools alternatively perform database or database-like searches for a single string or substring, called a query, in a collection of reads or references. In both cases, the mapping is one-to-one or one-to-many. The respective distributed memory parallelization is simple, because the reference or query can be replicated across processors and aligned to database or read set partitions independently in parallel. While these tools can technically be extended or used to solve many-to-many alignment problems in either shared or distributed memory, they tend to be inefficient, not having been designed or optimized for parallel many-to-many alignment. Many-to-many alignment is required for analysis and/or

[2]also called "multi-sequence" or "multi-way" alignment in the literature

assembly of reads from genomes for which references do not exist or are not reliable. Many-to-many alignment is much more challenging for distributed memory parallelization than many-to-one alignment is, because many reads must be aligned to many other reads in a pattern that cannot be determined until runtime, as it depends on the underlying (unknown) genome. A brute-force all-to-all approach to many-to-many alignment is feasible for very small data sets but quickly becomes intractable with increasing data set sizes. One of the foci and contributions of this work is a first scalable distributed memory many-to-many aligner, diBELLA[13].

## 2.3 Many-to-Many Long Read Overlap and Alignment as Hypergraph Construction and Refinement

The many-to-many long read overlap and alignment process can be understood in terms of the construction and refinement of an irregular hypergraph, in which reads are hyperedges and vertices contain information that is common to multiple reads and useful for alignment and refinement. We will use Figures 2.3-2.6 as running illustrations. The results of many-to-many alignment are useful for reconstructing the sequenced genome (*de novo* assembly). Also, before or in lieu of complete assembly, the results are useful for error-correction of the reads and direct analysis, with or without error-correction. In order to avoid the computational cost associated with computing all pairwise alignments directly (see Section 2.2), less expensive computations are employed to first detect similarity between reads. Similarity detection finds "candidate" read-to-read overlaps. Pairwise alignments are then computed only on these candidates to determine whether and how the reads overlap with one another. In other words, they construct an initial hypergraph that is refined through many-to-many pairwise alignment. Given the error rates of long reads, accurate similarity detection is not straightforward. Our approach computes and filters *k-mers* to both find candidate overlaps and the locations at which they are likely to overlap (see Figures 2.3-2.4) before alignment; the *k-mers* become seeds for seed-and-extend pairwise alignment. See Chapter 6 for related work. The formulation of the problem as hypergraph construction and refinement is not specific to *k-mer* based approaches, however; any read-to-read similarity computation can be substituted for *k-mer* matching in the following description.

Figure 2.3 illustrates a set of reads, numbered 1 to 4; the *k-mers* used to construct the hypergraph illustrated in Figure 2.5 are circled. The hyperedges in Figure 2.5 are reads. Hyperedge set intersections identify potential overlap between reads, and also alignment tasks for the refinement step. A pre-refinement *traversal* computes hyperedge set intersections. The refinement step computes associated pairwise alignments for the vertices in these intersections. If a resulting alignment score is relatively low, the respective vertex is pruned from the hypergraph. Otherwise, the alignment information is added to the information in the hypergraph. The ideal end-result is an overlap hypergraph in which exactly those reads


Figure 2.3: Example input reads, in no particular order, sharing circled 5-mers.

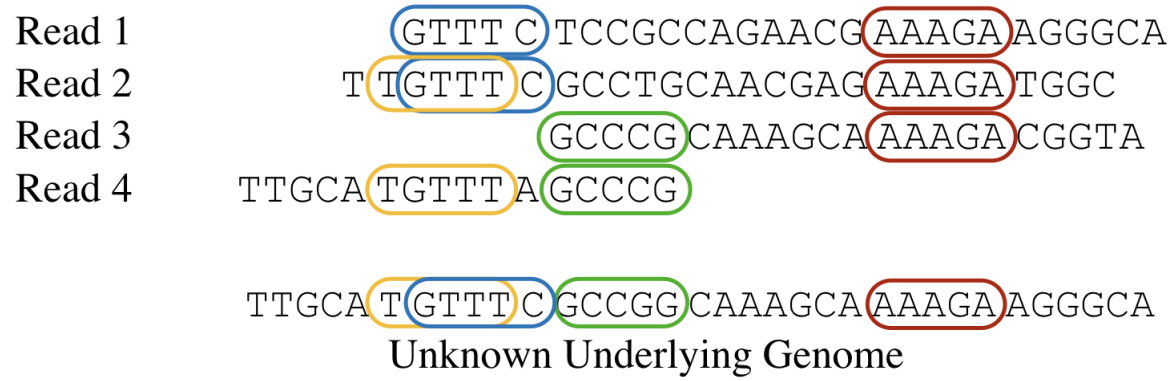


Figure 2.4: Reads and 5-mers from Figure 2.3 arranged according to the underlying unknown genome.

(hyperedges) that were sequenced from the same part of the genome (see Figure 2.4) have intersecting hyperedge sets, and furthermore, all vertices are labeled with pairwise alignment information for the respective pair of reads.

Figure 2.6 illustrates one of the challenges of distributed memory parallel refinement with two processors, $X$ and $Y$. In order to compute the hypergraph in Figure 2.5, suppose the four input reads in Figure 2.3 were split roughly evenly between the two processors as shown in Figure 2.6. After constructing the hypergraph and identifying hyperedge set intersections, we see that the processors each have one alignment task that can be computed entirely locally. The remaining 3 tasks each require 1 read from both processors. We can move the task to either process and roughly balance the number of tasks computed, but any way we divide the tasks, we will need to replicate and/or communicate reads between the processors.

While many alternative formulations are reasonable (see Appendix A), this formulation is convenient in discussing distributed memory parallelization challenges of many-to-many long read overlap and alignment. Distributed memory parallelization is particularly advantageous for processing workloads that do not fit in the memory of a single computer and for speeding-up the computation of many alignments (see Section 2.2 on the complexity of individual pairwise alignment). However, partitioning this hypergraph to balance the load

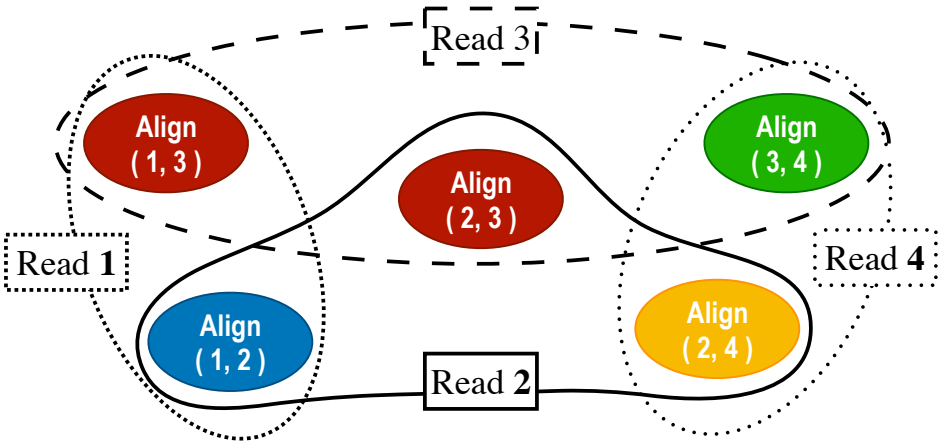


Figure 2.5: The hypergraph corresponding to Figure 2.3. Reads (hyperedges) are shown with solid and dashed lines connecting alignment tasks (vertices).

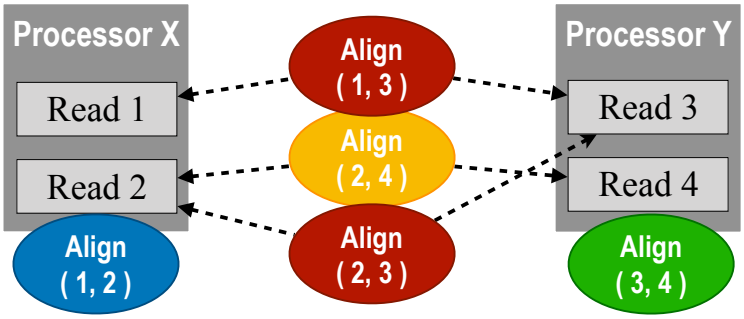


Figure 2.6: A possible partitioning of the read set in Fig. 2.3 between 2 processors, X and Y. Alignment tasks from Fig. 2.3- 2.5 are included.

is particularly challenging due to its irregular, data-dependent nature. In any partitioning that cuts hyperedges, reads will need to be duplicated and/or moved across partitions in order to compute pairwise alignments between reads in different partitions. In general, long reads are $[10^3, 10^5]$ characters in length, and the lengths vary widely within the same data set. Our running toy example shows 4 reads; the number of reads in real inputs depends on the underlying genome size $\Gamma$ and the sequencing coverage depth $d$. We will discuss these parameters in more depth in the following chapters.

## 2.4 *k-mer* Lengths and Frequency-Based *k-mer* Filtering

The right choice of *k-mer* lengths, $k$, for detecting similarity across reads (potential overlaps) in a given data set, balances the representation of distinctive genome characteristics with the need to detect similarities and/or eliminate errors across the data set. Maintaining distinctiveness (rare characteristic subsequences from the actual genome) tends to larger $k$'s, while error-tolerant similarity or error detection tends to shorter $k$'s (higher redundancies). However, if $k$ is too short, as an extreme, all reads may share all *k-mers*, even ones containing errors. In other words, it may be impossible to separate the signal from the noise. On the other hand, with a large enough $k$, all of the *k-mers* parsed from a data set may be unique, leading to a very uninteresting histogram, to say the least. If $k$ is large, it may be impossible to find exact matching substrings. The best choice of $k$ depends on the size of a given genome and the error rate of the input data set. Given the high error rates of long read data sets, $k$ tends to be $k \in [14, 17]$ for long read analyses. For short read single genome analyses, on the other hand, $k$ values tend to be much larger, for example $k = 51$. Models for choosing $k$ for long read overlap detection, relative to long read error rates, are supplied in our associated work on the Berkeley Long-Read to Long-Read Overlapper and Aligner (BELLA) [23].

Additionally, BELLA formalizes the concept of *reliable k-mers* for long read overlap detection. In short, *reliable k-mers* are probabilistically correct and unique length-$k$ subsequences in the underlying genome, according to their global frequency in the data set. Calculating the expected frequency of a unique *k-mer* depends on $k$ and the known coverage and error rate of the data set, which can be calculated from the input FASTQ file. As a simplifying example, in a world with error-free sequencing, the expected frequency of a *k-mer* that was unique in the genome would be equal to the sequencing coverage depth. The uniqueness property of *reliable k-mers* avoids incorrectly detecting overlaps between repetitive but otherwise unrelated regions of the genome. However, it may also fail to detect overlaps between reads that cover the same region of the genome, but do not include a subsequence that is unique in the genome overall. The distributed memory parallel pipeline, (di)BELLA relaxes the notion of *reliable k-mers*, and defines *retained k-mers* simply as those retained after a frequency-based filtering step. The frequency of retained *k-mers* is greater than one, as *k-mers* that occur only once cannot be used to detect overlaps between pairs of reads. The maximum frequency of a retained *k-mer* is a user-set parameter; the calculation for *reliable k-mers* in BELLA [23] can be used by default or an alternative supplied. The distinction is useful as we seek to support flexible range of analyses, as the technologies and models change and improve.

## 2.5 Scalability Analysis

Our empirical evaluations primarily focus on scaling behavior of the parallel algorithms and codes presented. To evaluate scalability, analytically and empirically, we frequently employ

two important metrics, *speedup* and *efficiency*. We also employ a variety of evaluation platforms for generality, and in order to study specific architectural trade-offs for our applications and workloads. Our workloads vary in size and sequencing characteristics, and all are used in real analysis of the respective genomes, rather than being synthetically generated.

## Strong Scaling and Weak Scaling Performance Analysis

In our empirical scalability analysis we frequently study *strong scaling* performance. That is, we study the performance of a code given a fixed workload with increasing parallelism, defined in terms of the number of parallel processors, $P$. We also study *weak scaling* performance but indirectly. Weak scaling studies increase the workload size relative to $P$. In this work, we analyze strong scaling performance across a number of (real) workloads which vary in size (see Table 2.4), but we do not examine weak scaling performance more directly. Speedup and efficiency, defined below, are therefore defined with strong scaling performance in mind.

## Speedup

Traditionally, *speedup*, $S$, is calculated as the serial time, $T_1$, over the time on $P$ parallel processors, $T_P$, to complete a computation. See Equation 2.1. Perfect strong scaling speedup is $S = P$.

$$S = \frac{T_1}{T_P} \tag{2.1}$$

Occasionally, slight variations of this definition are used in the following chapters; whenever the active definition departs from the traditional definition, that is specified in context.

## Efficiency

Efficiency is similar to speedup except that it measures the amount of work of each parallel processor with respect to the work by a serial processor running alone. See Equation 2.2.

$$S/P = \frac{T_1}{T_P} * \frac{1}{P} \tag{2.2}$$

In general, while speedup captures absolute improvement in running time due to parallelization, efficiency highlights the amount of overhead introduced by parallelization. Perfect strong scaling efficiency is $S/P = 1$. Super-linear speedup can be seen in an efficiency greater than 1.

## Architectures

Empirical performance analysis in this dissertation is conducted across a wide range of architectures for generality and to study varying balancing points between computing, memory,

| | **Cori II Cray XC40** | **Edison Cray XC30** | **Titan Cray XK7$^{\alpha}$** | **Genepool** | **Ethernet Cluster** |
|---|---|---|---|---|---|
| **Processor** | Intel Xeon-Phi (Knights Landing) | Intel Xeon (Ivy Bridge) | AMD Opteron 16-Core | Intel Xeon (Haswell) | AMD Opteron 8376 HE |
| **Freq (GHz)** | 1.4 GHz | 2.4 GHz | 2.2 GHz | 2.3 GHz | 2.3 GHz |
| **Cores/Node** | 68 cores | 24 cores | 16 cores | 32 cores | 32 cores |
| **Intranode LAT$^{\dagger *}$** | $3.3\mu$ | $0.8\mu$ | $1.1\mu$ | $2.7\mu$ | $0.6\mu$ |
| **BW/Node $^{\dagger \ddagger \S}$** | 57.3 MB/s | 436.2 MB/s | 99.2 MB/s | 113.0 MB/s | 1.2 MB/s |
| **Memory (GB)** | 96 GB | 64 GB | 32 GB | 256 GB | 512 GB |
| **Network & Topology** | Aries Dragonfly | Aries Dragonfly | Gemini 3D Torus | Infiniband Mellanox | Ethernet 1Gb & 10Gb |

Table 2.1: Evaluated platforms. *128 byte Get message latency in microseconds. $^{\dagger}$Using the optimal number of cores per node. ‡ Measured over approx. 2K cores or maximum (128 for Ethernet Cluster). § MB/s with 8K message sizes. $^{\alpha}$ CPU nodes only

and communication resources. Table 2.1 shows a number of the characteristics of the machines used in our studies. These include Cori, Edison, and Genepool at the National Energy Research Scientific Computing Center (NERSC)[3], an ethernet cluster at the Joint Genome Institute (JGI)[4], and Titan at Oak Ridge Leadership Computing Facility[5]. A few additional architectures, including an Amazon Web Services cluster, are introduced later in context.

## Workloads

Most of the empirical analysis presented throughout this dissertation is strong scaling performance analysis across the various (real) workloads presented in Table 2.4. For general analytical bounds, we capture important variables related workload specific characteristics; a reference list is supplied in table 2.2. These will also defined in the context in which they are used. Table 2.3 lists abbreviations and units commonly utilized throughout the dissertation.

[3]https://www.nersc.gov/systems/

[4]https://jgi.doe.gov

[5]https://www.olcf.ornl.gov/olcf-resources/compute-systems/

Table 2.2: Common variables used in equations related to workload sizes that depend on genomes, sequencing data, other domain specific characteristics, or the available parallelism.

| | |
|---|---|
| $P$ | number of parallel processors |
| $\Gamma$ | a variable for the size in base pairs or characters of a given genome |
| $k$ | the length of the character subsequences known as $k$-mers |
| $d$ | the average number of times a base pair in the genome is sequenced i.e. the coverage depth |
| $c$ or $C$ | with or without subscript, reserved for Big O notation of a context-defined constant |

Table 2.3: Reference table for common units and abbreviations.

| **Abbreviation/Unit** | **Definition** |
|---|---|
| bp or bps | nucleotide base pair (bp) or base pairs (bps) plural |
| byte | 8 bits |
| KB | 1 thousand bytes or $2^{10}$ bits |
| MB | 1 million bytes or $2^{20}$ bits |
| GB | 1 billion bytes or $2^{30}$ bits |

Table 2.4: Data sets used for evaluation. For each data set, we listed the name of the dataset, the scientific name of the species, the size of the `FASTQ` file containing the raw reads, and the source link.

| Short Name | Species | FASTQ Size | Source Link |
|---|---|---|---|
| *E. coli* 30× | *Escherichia coli* | 266 MB | https://bit.ly/2EEq3JM (CBCB) |
| *E. coli* 100× | *Escherichia coli* | 929 MB | https://bit.ly/2POV1Qs (NCBI) |
| *C.elegans* | *Caenorhabditis elegans* | 8.90 GB | https://bit.ly/2SU7Tqs (PacBio) |
| *Fruit Fly* | *Drosophila melanogaster* | 30GB | https://tinyurl.com/y9demqct (PacBio) |
| *Human* CCS | *Homo sapiens* | 25 GB | https://tinyurl.com/y73tfgnw (NCBI) |
| *Human* 10× | *Homo sapiens* | 62 GB | https://tinyurl.com/ya2eyrrc (PacBio) |
| *Human* 54× | *Homo sapiens* | 317 GB | https://tinyurl.com/yd2wyyln (PacBio) |

# Chapter 3

# Scalable Bulk-Synchronous Hash Table Construction for Irregular Hypergraphs

Managing memory, balancing load, and maximizing parallelism is especially challenging to do for irregular applications in a distributed setting. This chapter looks closely at two irregular and important applications from genomics, short read de Bruijn graph construction and long read overlap hypergraph construction. In particular, we examine the key design decisions and optimizations necessary for scaling these irregular applications within the arguably "regular" model of bulk-synchronous parallelism [61]. Since a distributed hash table, representing the respective hypergraph, is central to each application code, the patterns and optimizations we explore are likewise relevant to other irregular applications relying on distributed hash table construction. Our empirical analysis employs the code originally for de Bruijn graph construction within the extreme-scale *de novo* genome assembler, HipMer [20][18], and extended for long read overlap hypergraph construction by this author in diBELLA [13]. While our empirical analysis focuses on long read overlap hypergraph construction workloads, as the less well-explored of the two, the results are relevant to both applications, as we will additionally describe with analytical results. Among our observations, a common theme stands-out: the tight coupling between effective memory management and effective communication and computational load balance.

## 3.1 Introduction

This chapter examines scalable distributed hash table construction for two primary applications, short read de Bruijn graph construction and long read overlap hypergraph construction. Computing global *k-mer* histograms is a third general use-case, and while the first two use-cases compute *k-mer* histograms, the standalone use-case is not addressed in this work (see related work in Chapter 6). De Bruijn graph construction for short reads sampled from

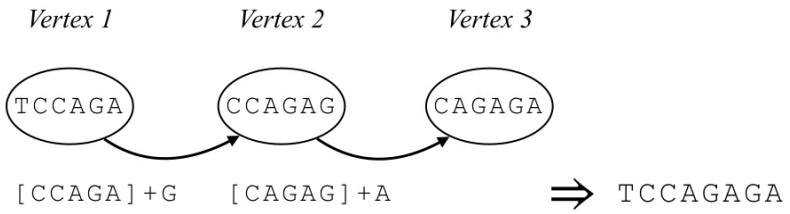


Figure 3.1: Toy example illustrating de Bruijn subgraph traversal with three 6-*mer* vertices. The path from Vertex 1 to 2 is computed by appending vertex 1's forward extension, 'G', to the last $k-1$ characters of vertex 1, "CCAGA". Likewise, the path from vertex 2 to 3 is computed by appending the forward extension, 'A', to the last $k-1$ characters of vertex 2, "CAGAG". Only forward traversal, and forward extensions with corresponding edges, are shown for simplicity. The result of the forward traversal is the string "TCCAGAGA".

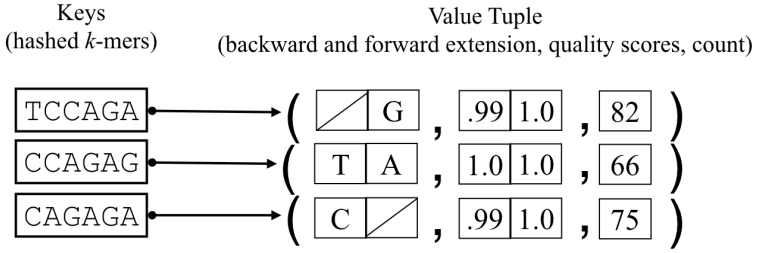


Figure 3.2: The global hash table for the toy example in Figure 3.1. The backward extensions, that were elided in Figure 3.1 for simplicity, are shown. The quality scores and counts shown were arbitrarily chosen.

a common genome involves parsing all fixed length substrings (*k-mers*) with their forward and backward extensions (single characters preceding and succeeding a given *k-mer* in each read). The *k-mers* are vertices in the graph. More appropriately, it is a de Bruijn (sub)graph because only the strings present in the data are represented, not all possible length-$k$ strings over the respective alphabet. The forward (or backward) extensions with the last (or first) $k-1$ characters of a given *k-mer* form an edge in the graph; Figure 3.1 illustrates. Filtering the *k-mers* based on frequency and the quality scores associated with each base pair (character) can be used to eliminate errors. HipMer stores this graph in a distributed hash table, in which the keys are *k-mers* and the values are the forward and backward extensions along with the quality scores. The global hash table for the toy example in Figure 3.1 is shown in Figure 3.2. In subsequent steps, traversal of this graph to find connected components produces sequences that are both error-free and longer than the original reads [18].

Long read overlap hypergraph construction, though semantically distinct (see Chapter 2, Section 2.3), follows a similar pattern. Approaches to long read overlap detection that are *k-mer*-based begin with parsing *k-mers* from long reads. Frequency-based filtering can

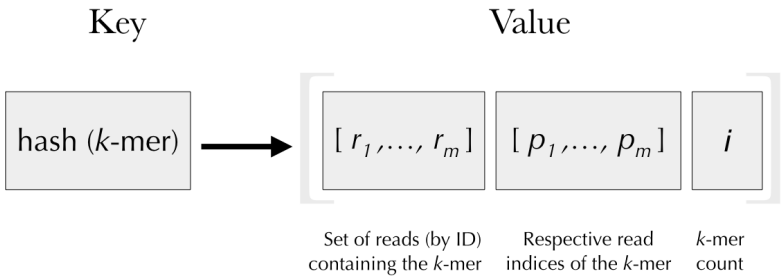


Figure 3.3: Diagram of the diBELLA's hash table elements. Read identifiers are denoted $r_1, ..., r_m$. *k-mer* positions are denoted $p_1, ..., p_m$. The global frequency of the respective *k-mer* is denoted $i$.

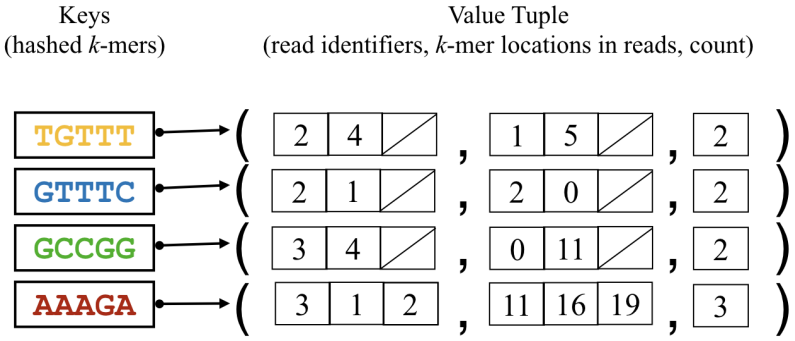


Figure 3.4: DiBELLA's global hash table for the toy example in Figures 2.3. Sets of read identifiers and corresponding *k-mer* locations stored in fixed-size arrays in the illustration.

be applied to the detection of *k-mers* that are probabilistically correct and unique in the underlying genome[23]. In turn, these filtered *k-mers*, or a superset of them, can be used to detect pairs of reads that are likely to overlap in the underlying genome. In an abstract representation in which *k-mers* are vertices in a hypergraph, reads are hyperedges that share multiple *k-mers* with multiple other reads. The intersections of hyperedge sets identify reads covering potentially overlapping regions of the genome. A distributed hash table is used to store this information in diBELLA[13]; the *k-mers* are hash table keys and the set of read locations in which the corresponding *k-mer* occurs are the values. A diagram of diBELLA's hash table elements is shown in Figure 3.3.

A global hash table for the toy example from Chapter 2, Section A.3, is shown in Figure 3.4.

The two applications share many computational patterns. Both parse and filter *k-mers* from input reads in order to construct a distributed hash table, compactly representing different kinds of graphs. In each application, *k-mer* frequencies supports *k-mer* analysis and filtering. The associated *k-mer* histogram produced is also useful in and of itself for genomic analysis. Both HipMer and diBELLA store the frequency of each *k-mer* as a value

in the hash table, along with the data described above, and optionally provide the histogram to the user.

The following sections review techniques for distributed memory scalability for these applications - see also [20][13]. We additionally extend the analytical results that underpin the success of certain techniques across these applications. We also present empirical results demonstrating the short-comings and trade-offs of certain techniques, not previously discussed in as much depth. Furthermore, the empirical analysis employs long read workloads, which are less well-studied and are even more irregular than short read workloads, for reasons explained in the context of each section.

At a higher level, there are two extremes for parallelizing this distributed hash table construction. One extreme is a bulk-synchronous parallelization that computes and communicates all *k-mers* and associated data in one large step. Another extreme is an asynchronous parallelization that overlaps the computation and communication of individual *k-mers* with associated data (one at a time). One parallelization approach minimizes the number of messages but maximizes memory usage; the other minimizes memory usage but maximizes the number of messages. This chapter examines an approach between these two extremes, a phased bulk-synchronous parallelization with multiple steps of computation and aggregated communication. The following chapters examine the other two extremes.

## 3.2 Distributed Memory Management of Input Reads

The lengths of input long reads are non-uniform and orders of magnitude longer than short reads. Long reads lengths vary between $10^3 - 10^5$ characters. Short reads are typically between $100 - 250$ characters, a narrower range for variability. In both short read and long read workloads, the total amount of data in the input depends on the underlying genome, size $\Gamma$, and the average number of times each base pair (bp) is sequenced i.e. the coverage depth, $d$. The input size, the result of sequencing a genome of size $\Gamma$ to coverage depth $d$, is $O(\Gamma \cdot d)$. Genome sizes vary widely, from 4.6 million bps of the *Escherichia coli* bacterial genome, to the 3 billion bps of the *Homo sapien* genome, to the 150 billion base pairs of the *Paris japonica* plant genome, to the sizes of so-called *metagenomes*, which are samples of microbial communities, that may be orders of magnitude larger.

To scale large and variable sized genomic workloads, HipMer and diBELLA distribute the input among $P$ parallel processors. The non-uniformity of input read lengths is managed by partitioning them by size in memory, such that each partition is as close to $O(\Gamma \cdot d/P)$ as possible - respecting read boundaries. In our empirical analyses, we have found that the parallel file I/O for reading the input, implemented with efficient C code, is a minimal and scalable runtime component. The algorithm and implementation is not so application-specific, nor is it a significant contribution to existing parallel I/O literature, that we dwell on it further.

In both applications, $k$ is strictly less than typical read lengths. Computing all *k-mers* for an input size of $O(\Gamma \cdot d)$ alone increases the working data set size in memory to $O(\Gamma \cdot d \cdot k)$.

With typical values of $k$ between 10 and 100 in both applications, this is **an increase of at least one order of magnitude** over the input. While the input data may fit in memory, for some configuration of $P$ parallel processors, the bag of all *k-mers* may not. Computing the (set) of all *k-mers* is the goal, however, and it is expected that the cardinality of the *k-mer* set is strictly smaller than the cardinality of the $O(\Gamma \cdot d)$ *k-mer* bag. We expect this due to the redundancy of the sequencing process, which in an error-free world, would lead to $d$ copies of every *k-mer* that exists in the genome. From observation, we also expect inherent redundancy in the genome. We cannot know the exact size of the *k-mer* set, relative to the size of the *k-mer* bag, until runtime. We can avoid storing the entire *k-mer* bag in memory at once though, by streaming the input while maintaining just the *k-mer* set. This is the first optimization for alleviating memory pressure that HipMer and diBELLA employ. The input is read in batches of roughly the same size, and the *k-mers* are computed and processed from those batches before the next batch is read. The *k-mer stream* referred to throughout the following is generated by this process. Streaming the input in batches serves not only to reduce memory requirement. It also has the beneficial side-effect of mitigating load imbalance from overall skew of the workload. We will return to this topic when discussing communication load balance in Section 3.3.

## 3.3 Load Balanced Hypergraph Construction

As a first step to hypergraph construction, *k-mers* are computed from input reads by moving a sliding window of length $k$ over each read 1 character at a time. From a read of length $L$, therefore, $L - k + 1$ *k-mers* can be parsed. Let $K_{bag}$ be the *k-mer* multiset ("bag" for short) computed from the input. The *k-mer* multiset cardinality, $|K_{bag}|$, for the input of size $\Gamma \cdot d$, is given in Equation 3.1. Equation 3.2 emphasizes the subtle point that, (in the long read case) where $L$ is three to four orders of magnitude larger than $k$, $(L - k + 1)/L \approx 1$. Note for clarity, the $O(\Gamma \cdot d)$ *k-mers* computed from the input (Equation 3.2) would consume $O(\Gamma \cdot d \cdot k)$ space if not streamed as described in Section 3.2.

$$|K_{bag}| = O(\frac{\Gamma \cdot d(L - k + 1)}{L}) \qquad (3.1) \qquad O(\frac{\Gamma \cdot d(L - k + 1)}{L}) \approx O(\Gamma \cdot d) \qquad (3.2)$$

Partitioning the reads in the input by size load balances the initial computation of *k-mers*. Parsing *k-mers* from read partitions is embarrassingly parallel – there are no dependencies between parsing a *k-mer* in one partition and parsing any other *k-mer* in any partition. However, reducing the *k-mer* bag to the corresponding *k-mer* set is a critical to constructing the global *k-mer* histograms, necessary for frequency-based *k-mer* filtering and constructing the final graphs. The distribution of *k-mers* in the input is unknown *a priori*, and, in

general,[1] there is no locality in the input files that we can rely on for a load balanced partitioning of the *k-mer* set. Hence, distinct *k-mers* are deterministically mapped to and redistributed to unique locations in a partitioned global hash table. The processor owning the respective partition is henceforward referred to as the *owner* of the *k-mers* that map to its partition. With a hash function that maps *k-mers* to processors uniformly at random, the growth in number of *k-mers* communicated (in aggregate) for each application is bounded by Formulas 3.3 and 3.4.

$$O(\frac{\Gamma \cdot d(L-k+1)}{L} \cdot \frac{P-1}{P}) \qquad (3.3) \qquad\qquad O(\Gamma \cdot d \cdot \frac{P-1}{P}) \qquad (3.4)$$

The bulk synchronous implementation redistributes *k-mers* in an irregular all-to-all exchange, implemented with MPI_Alltoallv. Given the expansion of the working data set size from the *k-mer* computation described in Section 3.2, completing the exchange in a single round may not be possible due to memory and runtime limitations. In such cases, the irregular all-to-all exchange is conducted in multiple rounds. Communication-batch sizes per round are limited by the same logic in each pipeline. Each parallel processor buffers data for remote processors until either of two conditions is met; (1) the amount of data buffered locally for any given target meets or exceeds a given constant $c_1$ or (2) the cumulative amount of data in all local buffers meets or exceeds another constant $c_2$. The values of constants $c_1$ and $c_2$ for the implementation were heuristically chosen and hand-tuned by the HipMer development team for Edison, but are presented here as architecture-dependent variables for generality. These imposed limitations not only avoid memory overflow and other undesirable behaviors. They also limit the impact of skew in the irregular all-to-all exchange; for example, per (1), the amount of data any processor receives from all other processors is limited to $(c_2 \cdot (P-1))$. We can lower bound the number of exchange iterations required at any scale using workload characteristics, including the size of *k-mers* in memory $M_{k\text{-}mer}$, and Formulas 3.5-3.6.

$$\Omega(\ (\frac{\Gamma \cdot d(L-k+1)}{L} \cdot \frac{P-1}{P} \cdot M_{k\text{-}mer})\ /\ (c_2 \cdot P)\ ) \qquad (3.5)$$

$$\Omega(\ (\Gamma \cdot d \cdot \frac{P-1}{P} \cdot M_{k\text{-}mer})/\ (c_2 \cdot P)\ ) \qquad (3.6)$$

The communication volume for constructing the distributed hashtable in each application depends on the representation of the metadata. Recall, the distributed hashtable represents a de Bruijn subgraph in HipMer. The metadata therefore includes the forward and backward extensions of each *k-mer*, single characters preceding and succeeding the *k-mer* in the read.

[1]With prior knowledge of the genome, some read reordering can improve locality. HipMer can, for example, use the the human reference genome for assembling new human genome samples. In Metahipmer, since the same input may be processed repeatedly for different values of $k$, information acquired on initial runs may be employed for subsequent runs.

These are necessary for traversing the de Bruijn subgraph. The quality scores corresponding to these characters, which can be though of as confidence weights, are also important metadata. In diBELLA, the hash table represents a hypergraph of reads connected by common *k-mers*, therefore the index of the source read is communicated along with each *k-mer* occurrence. The positions of shared *k-mers* between any two reads can be recomputed during alignment. However, this implementation records locations of probabilistically correct and globally rare *k-mers* by storing their positions along with the read index.

The scale at which both applications cross-over from being compute-bound to communication-bound for any given workload can be predicted with Equations 3.7-3.8, where $C_{kmer}$ denotes the computational cost of computing and hashing a *k-mer*, $C_{kmer} = O(k)$. The inverse bandwidth (for irregular all-to-all exchanges) is denoted $\beta$, and the startup cost or latency is denoted $\alpha$. The cross-over occurs when the computation cost is equal to the communication cost. After reviewing the Bloom filter optimization in Section 3.4, we will exercise these equations in Section 3.5 for two workloads, a de Bruijn and an overlap hypergraph construction workload, to make the discussion more concrete.

$$(\frac{1}{P})(\frac{\Gamma \cdot d \cdot (L-k+1)}{L} \cdot C_{k\text{-}mer}) \leq \ (\frac{P-1}{P})(\frac{\Gamma \cdot d \cdot (L-k+1)}{L})M_{kmer} \cdot \beta + \alpha \tag{3.7}$$

$$(\frac{\Gamma \cdot d}{P})\ C_{k\text{-}mer} \leq \ (\frac{P-1}{P})(\Gamma \cdot d)M_{kmer} \cdot \beta + \alpha \tag{3.8}$$

## 3.4 Minimizing Memory Requirement by Filtering-Out Singleton *k-mers*

The next optimization we discuss is filtering-out singleton *k-mers*, *k-mers* that occur only once in the entire input, to avoid the associated memory cost. For both long read overlap and de Bruijn subgraph construction, given a data set with sufficient coverage, singletons are likely the result of sequencing errors. For long read overlap hypergraph construction in particular, singleton *k-mers* are not useful for detecting overlaps between (pairs) of reads. Singletons have been observed to dominate short read workloads by $50 - 60\%$[18]. For long read workloads, singletons are potentially an even bigger problem, due to higher error rates and the relatively much smaller values of $k$. In a first streaming of the *k-mers*, both applications build a distributed Bloom filter to identify (with high probability) singleton *k-mers*, which need not be stored. This also enables the initialization of the distributed hash table with non-singleton *k-mers*. Briefly, a Bloom filter is a probabilistic data structure for space- and time-efficient set queries. It computes a number of hash functions for each element inserted and sets the respective bits in an array. For queries, if all bits for the queried element are set, the element may be in the set. Due to collisions, however a value may not be in the array even if its hash bits are set. If at least one bit is zero for a given

element it is guaranteed to be absent from the set [4]. In other words, a Bloom filter query may receive a false positives, but never a false negative. Section 3.6 discusses *k-mer* set cardinality estimation techniques for minimizing the false positive rate while minimizing the Bloom filter size and avoiding expensive resizing operations.

As mentioned, the input reads are distributed roughly uniformly by size over the processors using parallel I/O, but there are is no locality inherent in the input files. Each rank in parallel parses its reads into *k-mers*, hashes the *k-mers*, and eventually sends them to a processor indicated by the hash function. The hash function ensures that each rank is assigned roughly the same number of *k-mers*. The algorithm proceeds in two logical steps, the first exchanges *k-mers* (without metadata) to filter-out singleton *k-mers* and initialize the hash table keys. The second logical step initializes hash table values, exchanging both *k-mers* and metadata. In the first step, received *k-mers* are inserted into the local Bloom filter partition on the destination processor. If a *k-mer* was already present, it is also inserted into the local hash table partition as a key. Although all $O(\Gamma \cdot d)$ *k-mers* are to be computed, each step is performed in substages since only a subset of *k-mers* may fit in memory at one time, as discussed in Section 3.2. The Bloom filter construction communicates nearly all (roughly $(P-1)/P$) of the *k-mer* instances to other processors in a series of bulk synchronous phases. The total number of phases depends on the size of the input, as described in Section 3.3. The only difference between the communication for initializing the Bloom filter (step one) and initializing the hash table (step two) is that only the *k-mers*, and not also the metadata (e.g. source read identifiers for overlap hypergraph construction or forward and backward extensions for de Bruijn subgraph construction), are communicated for initializing the Bloom filter. The second step, initializing hash table keys, communicates all *k-mers* (with metadata) in substages again. The destination processor only stores the received metadata for *k-mer* keys already in the hash table. Empirical results for both the Bloom filter and hash table initialization are presented together in Section 3.8, following the discussion of optimizations in the next sections. Finally, after the hash table is initialized with *k-mer* keys (for *k-mers* that occur more than once), the Bloom filter is freed.

## 3.5 Analytical & Empirical Results for a Long Read and a Short Read Workload

Let us exercise the analytical bounds from Section 3.3 to understand the impact of workload characteristics on the performance of HipMer and diBELLA's respective hash table constructions. We employ two (real) workloads as examples, one short read Illumina and one PacBio long read human data set. For each, $\Gamma \approx 3 \times 10^9$. For processing long reads with 15% error, diBELLA employs $k = 17$; whereas for short reads with much less than 1% error, HipMer employs $k = 51$, a value tuned by domain experts. The diBELLA and HipMer codes represent *k-mers* with the same 2-bit encoding objects. The size of *k-mers* in memory at compile time and runtime depends on $k$; diBELLA stores its 17-mers in 64bits and HipMer

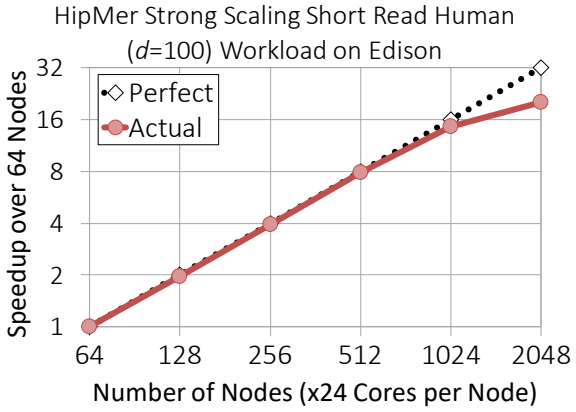


Figure 3.5: Strong scaling speedup of HipMer's de Bruijn graph construction given an Illumina short read human data set with 100× coverage. Speedup (y-axis) is calculated relative to performance on the minimum number of nodes required to process the data set (64). The number of nodes is shown on the x-axis. MPI ranks are pinned to cores (24 ranks per node).

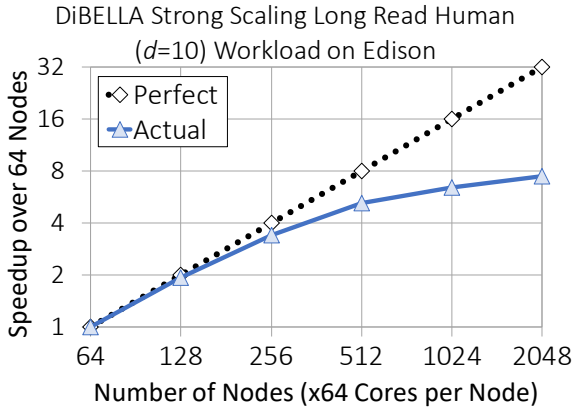


Figure 3.6: Speedup of diBELLA's overlap hypergraph construction on Edison, given a PacBio long read human data set with 10× coverage. Speedup (y-axis) is calculated relative to the performance on 64 nodes. MPI ranks are pinned to cores (24 ranks per node). The number of nodes is shown on the x-axis.

stores its 51-mers in 128bits. The average and median long read lengths of the PacBio data set are roughly 7,400 and 6,300 bps, respectively; $(6300 - 17 + 1)/6300 \approx 1$, so Equation 3.2 checks-out. The Illumina short reads are 101 bps, so roughly $k$ 51-mers are computed from each short read. A last important difference between the two data sets is the coverage, $d$. High-coverage short read data sets are available for the human genome, as evidenced by this data set with $d \approx 100$. The long read data set on the other hand has coverage $d = 10$. We will look at the results maintaining this difference, and then discuss the impact of a fixed a coverage between the two.

Let us plug the values of $\Gamma$, $k$, $L$, and $d$ into Equations 3.3 and 3.4, respectively, to calculate the expected, relative differences in load. Given a 10× difference in coverage $d$, the result is that HipMer computes and communicates over 5× as many *k-mers* as the diBELLA run (roughly 150 billion versus 30 billion). In the irregular all-to-all *k-mer* exchanges to initialize the distributed Bloom Filter (which communicate only *k-mers* and no metadata), this translates to 10× difference in aggregate communication volume, due to the 2× difference in *k-mer* representations (128 bits for HipMer's 51-mers versus 64 bits for diBELLA's 17-mers). HipMer's representation of *k-mers* (with metadata) is 1.1× the size of diBELLA's representation of *k-mers* (with metadata). HipMer's aggregate communication volume for initializing the hash table with metadata is almost 6× higher than diBELLA's.

The impact of the 10× difference in coverage on the communication volumes is as follows.

If we were to fix $d = 10$ for both workloads, diBELLA would compute and communicate $2\times$ as many *k-mers* as HipMer (roughly 30 billion versus 15 billion). Given that HipMer's *k-mer* representation is $2\times$ larger than diBELLA's, their communication volumes for Bloom filter initialization would be roughly equal. For initializing the distributed hash table (with metadata), given that Hipmer's reprentation is only $1.1\times$ larger than diBELLA's for this, diBELLA's communication volume would be $1.8\times$ higher than HipMer's.

Returning to the real workload characteristics, we can estimate the number of exchange iterations required at any scale using the aggregate communication volume (Equations 3.3-3.4), the runtime settings for Edison, $c_1 = 2$MB and $c_2 = 128$MB, and Equations 3.5-3.6. As we strong scale, the number of iterations needed to communicate the same aggregate amount of data decreases inversely in both applications. With 1,024 nodes (24,576 parallel processors) and up, HipMer's graph construction can complete the exchange, for initializing the distributed hash table, in a single iteration. For diBELLA's workload, at least 256 nodes (6,144 parallel processors) are required to do the same according to the analytical estimate. Empirically, two exchange iterations are required at the 256 node scale. The second exchanges a small fraction of the overall communication volume and the first is under the limit, implying that some processors met condition (1) with $c_1$ in the first exchange.

Strong scaling results for the two workloads are presented in Figures 3.5 and 3.6. The scale at which both applications cross-over from being compute-bound to communication-bound for the respective workloads is predicted with Equations 3.7-3.8. Given the $5\times$ difference in the computational load, HipMer does not cross-over until $1,024$ nodes for this workload, whereas diBELLA crosses-over around 256 nodes.

## 3.6 Precise Cardinality Estimation Trade-Offs

This section examines the memory-computation trade-off of precise *k-mer* set cardinality estimation within our two applications. Our case in point is the near-optimal set cardinality estimator, HyperLogLog [25], implemented in both HipMer [18] and diBELLA [13]. We discuss and compare the memory and computational costs of this algorithm against a simple heuristic also implemented in both codes. First, we review the purpose of *k-mer* set cardinality estimation in the context of these applications and discuss the pertinent details of each estimator.

Estimating the cardinality of the *k-mer* set is primarily used for initializing the Bloom filter described in Sections 3.4. The cardinality is used to determine the Bloom filter size and number of hash function that will achieve the desired false-positive rate, while avoiding expensive resizing operations. For the hash table, described in Section 3.3, knowing the set cardinality is convenient but not critical. The hash table will contain the subset of *k-mers* filtered by the Bloom filter. Rather than initializing the hash table size directly to the estimated set cardinality, the actual code works with available memory and other methods for handling hash table collisions.

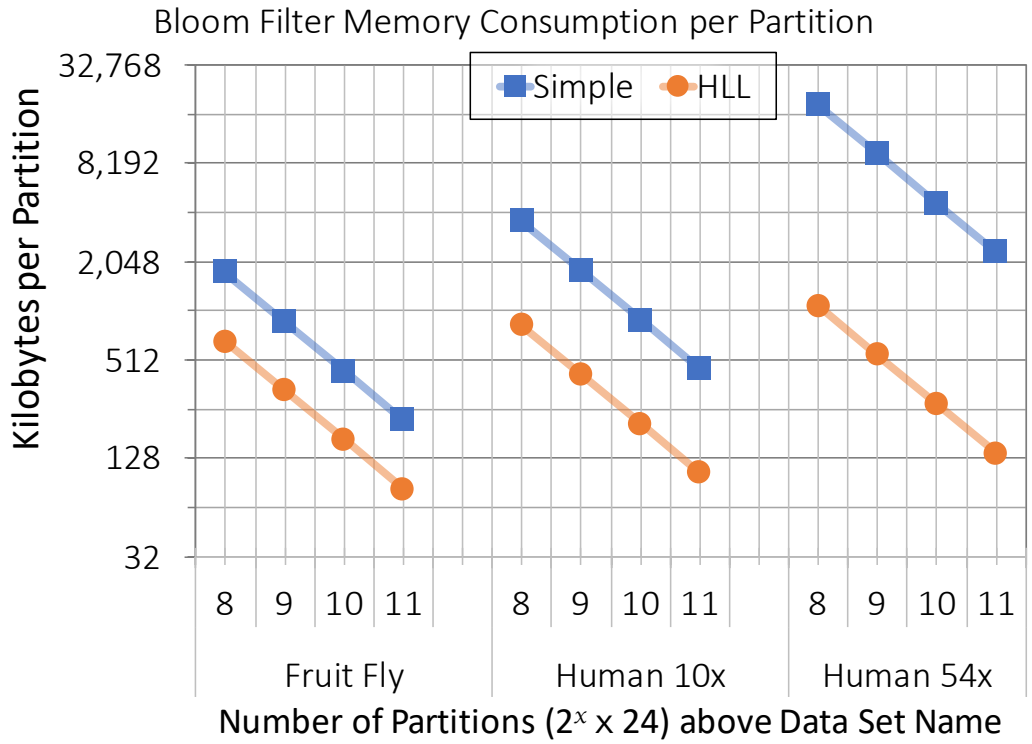


Figure 3.7: Comparison of empirical Bloom filter sizes initialized with the "Simple" versus the HyperLogLog ("HLL") cardinality estimation techniques. The comparison is across 3 data sets and 4 scales each (x-axis). The number of partitions is 2 raised to the power of the value on the x-axis times 24 – 1 partition per Edison core was used in experiments, Edison has 24 cores per node.

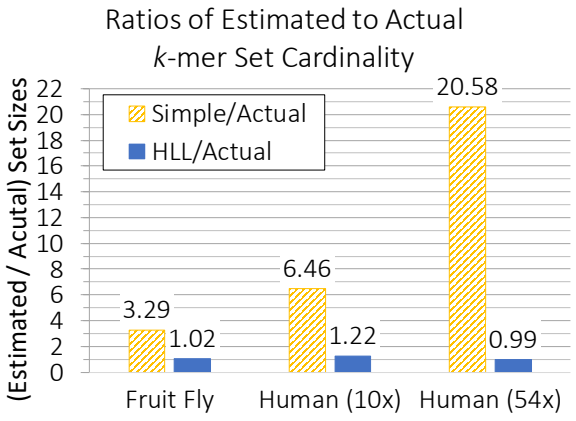


Figure 3.8: Compares ratios of the estimated to the actual *k-mer* set cardinality for each estimation approach, the "Simple" and the HyperLogLog ("HLL"), across 3 data sets.

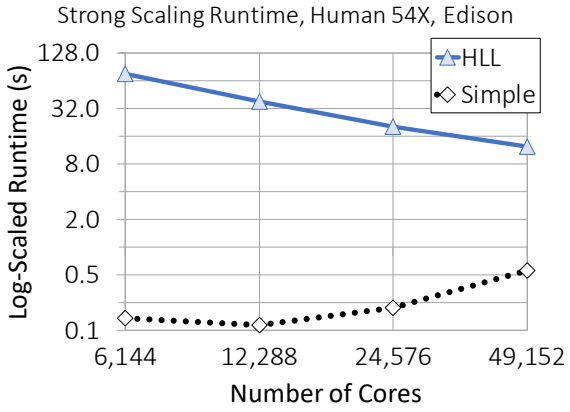


Figure 3.9: Compares the (log-scaled) runtime of each estimation technique, the "Simple" and the HyperLogLog ("HLL"), strong scaling with the *Human* 54× data set on Edison.

Therefore, cardinality estimation is not strictly necessary for ensuring correctness of the main algorithm. Filtering-out singleton *k-mers* with the Bloom filter is a memory-saving optimization. Avoiding storing singleton *k-mers* this way offers significant memory savings in practice, but otherwise it is not critical to the correctness of the main algorithm. Further, Bloom filters only report false positives and never false negatives; in our context, this means that independent of the cardinality estimation used to initialize the Bloom filter, non-singleton *k-mers* are never mistakenly filtered-out. False positives in our context are singleton *k-mers* that collide with some other *k-mer* in the Bloom filter, and therefore appear to the Bloom filter to occur more than once in the data set. Accurately estimated (or overestimated) cardinality is an alternative to expensive resizing operations for minimizing the Bloom filter's false positive rate. In both applications, however, there is additional handling in place for eliminating Bloom filter false positives. Once the hash table is constructed and initialized with *k-mers* that passed the Bloom filter, each processor independently in parallel traverses its hash table partition to eliminate singletons that "fooled" the Bloom filter. For diBELLA in particular, this step adds little to the $O(\Gamma \cdot d/P)$ parallel traversal that is performed anyway, in order to remove *k-mers* that occur more frequently than a user-defined threshold[2]. In short, cardinality estimation is not strictly necessary for ensuring correctness either of the main algorithm, that relies on the global hash table, or of the Bloom filter optimization.

The merits precise cardinality estimation are that it can avoid expensive resizing operations, while minimizing the size of the Bloom filter, compared to high overestimation. The HyperLogLog algorithm is a proven near-optimal cardinality estimate for sets that can be represented numerically [25]. In the context of our applications, a numerical representation for *k-mers* is derived from a hash function. Our alternative estimator, used as a baseline for comparison, is a heuristic that estimates the *k-mer* set cardinality to be directly proportional to the *k-mer* bag size. Thus this simple heuristic overestimates the *k-mer* set cardinality in practice and supports a minimal Bloom filter false-positive rate without resizing. A key difference between the two estimation techniques is their relative complexity. The simple heuristic requires only the input file size to estimate the *k-mer* bag size. With this, the calculation can be done independently in parallel, and is essentially constant-time. HyperLogLog on the other hand requires the maximal element, the most frequently occurring element.[3] Finding the maximal element requires another streaming of the input *k-mers*, an $O(\Gamma \cdot d/P)$ computation in addition to the communication costs.

Figures 3.7-3.9 compare the relative space costs, runtime overhead, and accuracy of the two cardinality estimators across three real long read workloads, *Fruit Fly*, *Human* 10×, and *Human* 54×. Figure 3.8 shows a comparison of the accuracy achieved with HyperLogLog ("HLL") versus the "Simple" heuristic for these three workloads. Cardinality estimation accuracy is calculated as the ratio of the estimated to the actual *k-mer* set cardinality of the given data set for $k = 17$, our default $k$ for long read analysis[23]. Across data sets, the

[2]As described elsewhere, these can be considered "noise" for the purpose of detecting *k-mers* that are unique in the genome (detecting reads that overlap along unique regions of the genome).

[3]More precisely, it requires the number of leading zeros in the binary representation of the maximal element; conditions, caveats, and full details are provided in the associated publication [25].

HyperLogLog ("HLL") estimate is within 22% of the true set cardinality. It is only off by 2% for the *Human* 10× workload. For the *Human* 54× data set, it slightly underestimates the cardinality, leading to a higher false-positive Bloom filter rate than the overestimated cardinality. This leads to more *k-mers* being inserted into and then removed from the hash table afterward than otherwise. However, given that the estimate is still within 1% of the true cardinality, the overhead of this is negligible. The simple heuristic based on the *k-mer* bag sizes (labeled "Simple" in the figures), overestimates the cardinality in all cases, by a factor of roughly $\approx 3\times$ for the *Fruit Fly* workload, up to a factor of $\approx 20\times$ for the *Human* 54× workload.

While HyperLogLog clearly offers higher accuracy in these results, we also examine the runtime cost in practice. Figure 3.9 shows the performance of each estimator for the largest data set, *Human* 54×, strong scaling across $256 - 2048$ nodes (with 24 cores per node) on Edison. As expected, the $O(\Gamma \cdot d/P)$ HyperLogLog algorithm ("HLL" in the figure) takes as long as any other phase of the main algorithm that streams all the *k-mers*. It also scales similarly to those; at first, it scales linearly from $2566 - 512$ nodes $(6,144 - 12,288$ cores), but the speedup gradually tapers off between $1024 - 2048$ nodes $(24,576 - 49,152$ cores). The "Simple" heuristic on the other hand takes less than $0.5s$ across scales, a negligible amount of time relative to the rest of the application. The tail increase from roughly $0.1s$ to $0.5s$ is presumably due to file system inefficiencies, in supporting $24,576 - 49,152$ concurrent readers across $1,024 - 2048$ nodes. That it increases so little for these concurrency levels is a virtue of the HPC file system.

Finally we examine relative space costs across the three workloads. Figure 3.7 shows the absolute size in memory of the Bloom filter partitions (initialized using each estimate). Just as one would infer from the results in Figure 3.8, the Bloom filters initialized with the "Simple" estimate are much larger than those initialized with the HyperLogLog ("HLL") estimate. However, the absolute space consumption is small. For example, Bloom filter partitions, initialized with the simple estimate for the *Fruit Fly* workload, are 2MB. On the architecture on which these results were collected, Edison's Intel Xeon cores, 2MB is 0.07% of the per core memory. Even on a hypothetical 1GB/core architecture, that is 0.1% of the per core memory. For the largest example workload, *Human* 54×, Bloom filter partitions initialized using the simple heuristic are at most 16MB – 0.6% of the per core memory on Edison. When strong scaling, all data structure partition sizes, including the Bloom filter partition sizes, decrease linearly with the total number of partitions.

In summary, the HyperLogLog optimization provides highly accurate cardinality estimation both in theory and for our demonstration workloads (Figure 3.8). However, given the runtime overhead, its necessity is debatable given the space savings, except for extremely small ratios of memory to computational resources. Bloom filters, as a space efficient data structure by design, offer a small memory footprint even for highly overestimated set cardinalities, as demonstrated for the workloads in Figure 3.7. Furthermore, the memory cost is distributed across processor partitions and scales. The computational overhead (essentially another streaming of the *k-mers*) is far from free (see Figure 3.9) when not combined with another essential streaming of the *k-mers*. In Hipmer[18], the streaming of the *k-mers* for

cardinality estimation is combined with that of the heavy hitters optimization. The next section reviews the heavy hitters optimization, its necessity and effectiveness. Particularly, the next section examines cases in which the heavy hitters optimization cost outweighs the associated benefits, and is therefore relevant to the current discussion on precise cardinality estimation as well.

## 3.7 Heavy Hitters Detection Trade-Offs

The distribution of *k-mers* in the input is not known until runtime, and moreover, can be extremely skewed. Borrowing a term originating from the database literature, *k-mers* that occur extremely frequently in an input stream are referred to as *heavy hitters*. Heavy hitters are generally problematic in networks and distributed systems that use identity mapping to distribute load.

One highly scalable solution to heavy-hitter *k-mers* was proposed by Georganas et al. and implemented in HipMer for short read analysis [20]. Their optimization detects heavy hitters with a high performance implementation [6] of the Misra-Gries counting algorithm [46]. The optimization in HipMer computes *k-mers* in an initial streaming of the input, in order to detect heavy hitters. In a second streaming of the input, following the detection phase, the algorithm does not immediately send heavy hitters to their respective owner(s). Rather, the source(s) compute a partial sum of each heavy hitter, and send both the *k-mer* and its count (once) to the respective owner. The optimization essentially redistributes the computational load of counting high frequency *k-mers* among the owner and the sources, and it avoids hammering the network with those *k-mers*. In other words, it adjusts load imbalance in both the computation and in the irregular all-to-all communication from heavy hitters. Georganas et al. demonstrated the efficacy of this approach for extremely skewed workloads, including the hexaploid bread wheat lines, "Synthetic W7984"[20].

Among other factors, heavy-hitter *k-mers* arise due to the relationship of $k$ to the length and frequency of inherently redundant genomic sequences. For long read workloads, unlike short read workloads, $k$ is extremely small relative to average read lengths. Between two hypothetical short and long read workloads, sequenced from the same genome with the same coverage depth, it is therefore possible for the long read workload to contain significantly more high-frequency *k-mers*. In this section, we compare the overhead of the heavy hitters optimization to the empirical performance benefit of its load balancing effects on two long read workloads. We show that the benefit of this optimization does not always outweigh the cost. As part of the study, we use essentially the same optimized and unoptimized implementations as Georganas et al. [20] [18]. We also use the same experimental setup with Edison at NERSC, except for that we scale to larger numbers of nodes in powers of 2 (up to $2,048$ versus 640 nodes). In all experiments, each node is populated with 24 MPI ranks, 1 pinned to each core. The experiments are scaled out to $49,152$ cores. Figures 3.10-3.17 provide strong scaling results from two example long read workloads in which the overhead of the optimization ultimately obviates its improvement to overall performance.

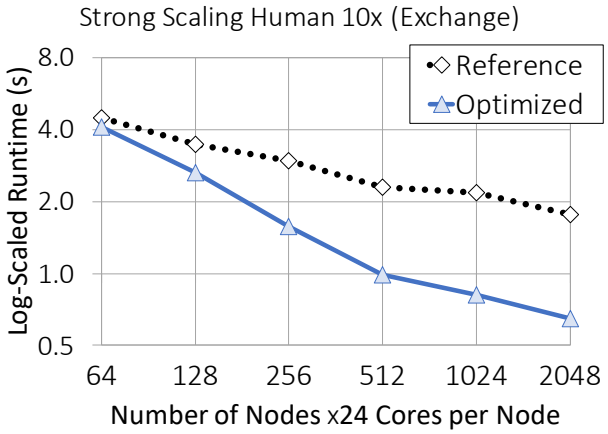


Figure 3.10: Strong scaling comparison of the irregular all-to-all *k-mer* exchange time. (Lower is better.) The heavy-hitters optimization improves the exchange performance and scalability.

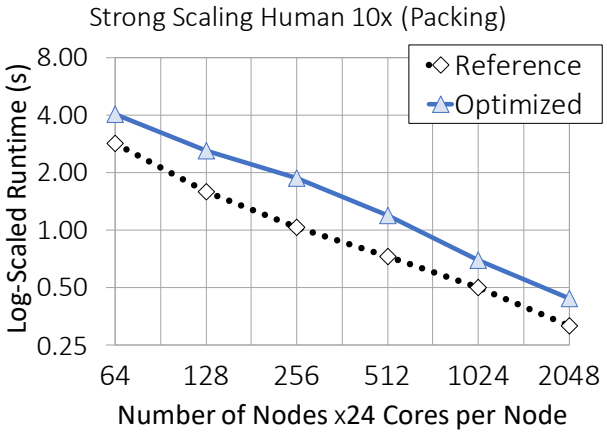


Figure 3.11: Strong scaling comparison of time spent packing *k-mers* for the irregular all-to-all exchange. The heavy-hitters optimization requires more packing time to consolidate heavy-hitters at the source(s).

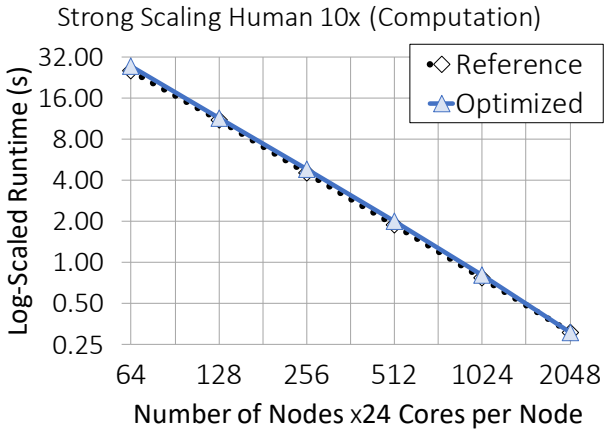


Figure 3.12: Performing partial counts of heavy hitters at their sources in the optimized version increases the computation time negligibly $(0 - 7\%)$ over the reference version.

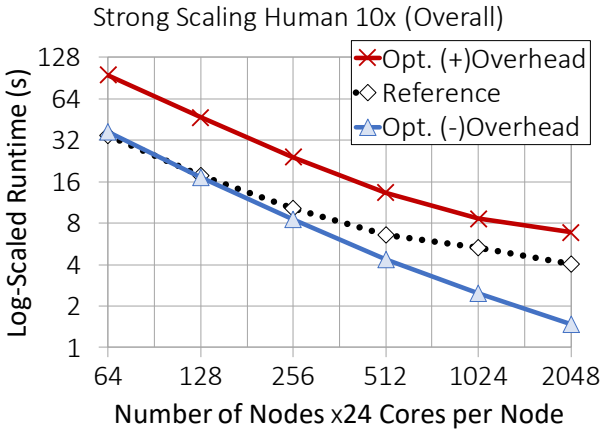


Figure 3.13: Shows the overall heavy-hitters optimized performance with $(+)$ and without $(-)$ the cost of the optimization itself, alongside the overall performance of the reference implementation.

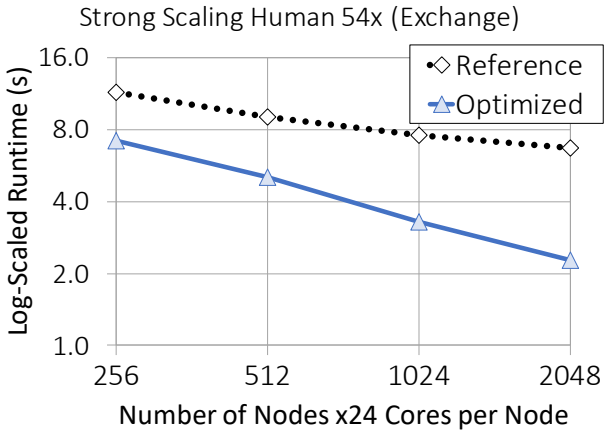


Figure 3.14: Strong scaling comparison of the irregular all-to-all *k-mer* exchange time. (Lower is better.) The heavy-hitters optimization improves the exchange performance and scalability.

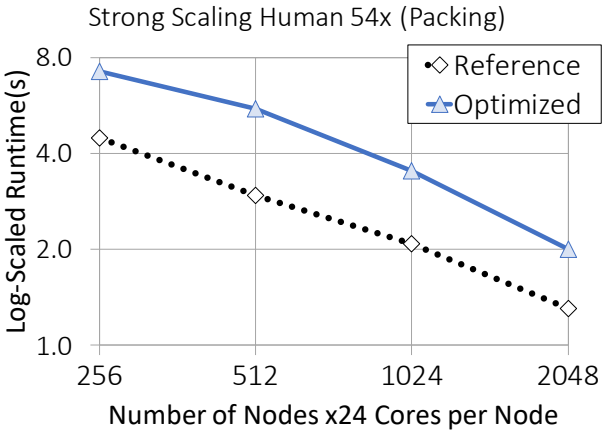


Figure 3.15: Strong scaling comparison of time spent packing *k-mers* for the irregular all-to-all exchange. The heavy-hitters optimization requires more packing time to consolidate heavy-hitters at their sources.

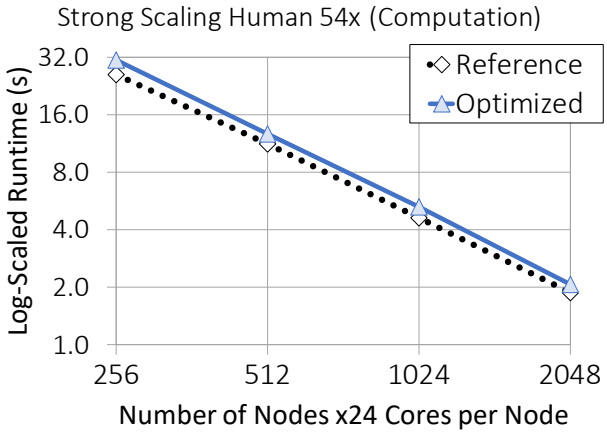


Figure 3.16: Performing partial counts of heavy hitters at their sources in the optimized version increases the computation 10-20% over the reference version.

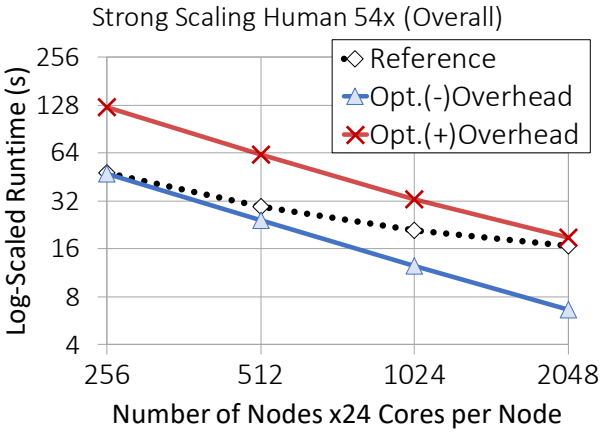


Figure 3.17: Shows the overall heavy-hitters optimized performance with (+) and without (−) the cost of the optimization itself, alongside the overall performance of the reference implementation.

Figures 3.10-3.13 show the results of strong scaling hash table construction, described in Section 3.3, for the *Human* 10× data set with $64 - 2048$ Edison nodes ($1,536 - 49,152$ cores). The performance of the code labeled "Reference" is from running the code without the heavy-hitters optimization; the *k-mers* are not streamed to detect heavy hitters and there is no consolidation before communication of any of the *k-mers*. The effects of the heavy-hitters load balancing optimization on exchange, packing, and computation time are apparent. As shown in Figure 3.10, both the absolute runtime and the scalability of the irregular all-to-all *k-mer* exchange improves. With 64 nodes (1,536 cores) the improvement over the *Reference* version is only 0.4×, but unlike the *Reference* exchange, it scales linearly to 512 nodes (12,288 cores). With $1024 - 2048$ nodes, the improvement over the *Reference* is $\approx 2.7\times$. However, turning now to Figure 3.11, the optimized version spends more time packing than the *Reference* version. This is expected, since the optimized version additionally traverses its local data structure of heavy hitters to pack them. Still, at any scale, neither version spends a significant portion of the overall runtime in packing. Figure 3.12 shows the strong scaling computation time of each version. The computation time in the optimized version includes time to compute correct partial sums of heavy hitters; the initial pass, executing the Misra-Gries algorithm[46] on the *k-mer* stream, only guarantees the identity and lower bound on the frequency of heavy hitters. The overhead of this computation in the optimized version, however, is negligible. The computation of both implementations speeds-up super-linearly, as we observed in all other *k-mer* analysis passes. The results labeled "Opt. (-)Overhead" (blue triangle-marked line) in Figure 3.13 show how the improvement in the exchange time and scalability, the increase in packing time, and the slight increase in computation time balance out in the overall performance (not including the overhead of the optimization itself). The reference performance is marked "Reference" in the same figure. The scalability improvement to these components over the reference is clear. Figure 3.13 also shows the overall performance (computation, packing, exchange) combined with the overhead of the optimization itself (see "Opt. (+)Overhead", the red ×-marked line). While the optimized version scales better than the reference, the runtime is **roughly double that of the *Reference* across scales** due to the additional streaming of all *k-mers*. The optimized version speeds-up perfectly to 256 nodes, whereas the reference speeds-up perfectly only to 128 nodes. The reference is 1.7× faster than the optimized version with 2048 nodes.

The second workload in our analysis, *Human* 54×, is sequenced from the same genome with much higher sequencing coverage. The input size alone is over $300GB$, an over 5× increase over the previous workload. From both the increase in size and redundancy from higher sequencing coverage, we expect heavy hitters to be a larger problem for this workload than for the *Human* 10× workload. As we will see, however, the heavy hitters optimization does not quite payoff. Figures 3.14- 3.17 show hash table construction strong scaling results with $256-2,048$ Edison nodes ($6,144-49,152$ cores). Again, the performance and scalability of the irregular all-to-all *k-mer* exchange is improved significantly, as shown in Figure 3.14. The cost of packing more than doubles however (Figure 3.15). The increase in computation is also more apparent with this workload (Figure 3.16); it increases by 20% with 256 nodes but drops down to 10% with 2048 nodes. The speedup in the computation of both versions

is superlinear from $256 - 512$ nodes ($8,704 - 12,288$ cores). From $1,024 - 2,048$ nodes ($24,576$-$49,152$ cores) however, the speedup very gradually tapers-off. Let us examine how these competing effects balance-out in overall performance (Figure 3.17). The optimization improves the overall runtime and scalability of hash table construction (see the blue triangle-marked line, labeled "Opt.(-)Overhead") over the reference (black diamond-marked line). However, the cost of the optimization combined with the improved performance (labeled "Opt.(+)Overhead") is more than the runtime of the reference version. Only at the 2048 node scale does the optimized version come close to closing the gap between it and the *Reference*.

From the results presented elsewhere for HipMer[20][18], and in Figures 3.10 and 3.14, we learn that the heavy hitters optimization is particularly effective for improving the scalability of the irregular all-to-all *k-mer* exchanges. Georganas notes this optimization is essentially free when combined with the cardinality estimation [18] discussed in Section 3.6. This is because the heavy hitters optimization keeps track of the most frequently occurring (set) of elements (*k-mers*) per partition, while the HyperLogLog cardinality estimation only requires the single most frequently occurring element. In other words, in scenarios in which the HyperLogLog cardinality estimation is worthwhile, or the heavy-hitters optimization is worthwhile, the overhead of running both instead of just one is negligible. However, as discussed in this and the previous section, the benefit of either optimization may not outweigh the respective cost. From the other results presented in this section, we learn that unless (any) additional streaming of all *k-mers* is necessary to the application, the overhead is not mitigated by the performance and scalability improvements for every workload. We remark for future work, however, that there is clear potential to minimize this overhead with probabilistic algorithms that do not sample the entire *k-mer* data set.

## 3.8 Overall Performance

The final set of results we present is the combined strong scaling performance of the hash table and Bloom filter construction (Sections 3.3-3.4) from diBELLA for the long read *Human* $10\times$ data set. The experiment excludes those optimizations described and shown to be unnecessary for this workload in Sections 3.6-3.7, namely the HyperLogLog cardinality estimation and the heavy hitters optimization. Figure 3.18 shows the runtime broken down into communication and computation for constructing each data structure. The computation for both, which includes computing and hashing all *k-mers*, speeds-up linearly. The runtime switches from being computation bound to communication bound at $512 - 1024$ nodes ($12,288 - 24,576$ cores) – see Equation 3.8. The absolute runtime drops from just over a minute with 64 nodes to just over 5s with 1024 nodes.

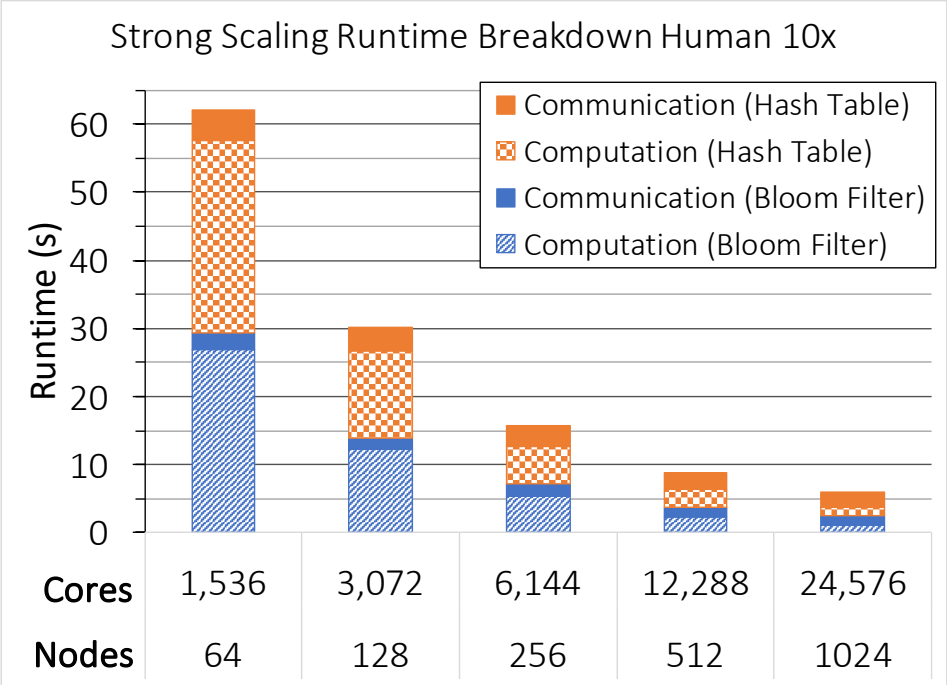


Figure 3.18: Hash table and Bloom filter construction runtime breakdown, strong scaling with the *Human* 10× data set on Edison.

## 3.9 Conclusions

In this chapter, we have closely examined several techniques for "regularizing" two irregular applications, de Bruijn graph construction in HipMer [18] and overlap hypergraph construction in diBELLA [13], for distributed bulk synchronous parallelism. By "regularizing" we simply mean, forcing the computational pattern into uniform computational steps separated by uniform communication steps according to the original BSP model [61] – as nearly as possible, that is, given the irregularity of our applications. Sources of irregularity in these applications arise from variable genome sizes, error rates, unknown *a priori* coverage and read set cardinality, non-uniform and unknown distributions of read lengths, at-runtime knowledge of $k$ and unknown numbers and distributions of *k-mers*. These irregularities present themselves more extremely in long read overlap hypergraph construction, in which reads vary between $10^3 - 10^5$ (versus $100 - 250$ short read) character lengths, and error rates historically have been between $5 - 35\%$ (versus below $1\%$ for short reads). As such and because it is the less well-studied problem, we focused our empirical analysis on long read overlap hypergraph construction.

As observed, design decisions for effective memory management, given such irregularity, also lead to improved performance and load balance in both the computation and communication. Partitioning non-uniform reads by size not only balanced memory (Section 3.2) but also resulted in linear *k-mer* computation speedups due to the relationships described in

Equations 3.1-3.2 and 3.7-3.8 (see Section 3.3 and Figures 3.12, 3.16, 3.18). Processing reads and *k-mers* in batches not only supported workload processing within limited amounts of memory (Section 3.3), but also had the beneficial side-effect of limiting the impact of overall workload skew on irregular all-to-all *k-mer* exchanges.

While we examined several techniques such as these in which memory management and load imbalance design decisions have gone hand-in-hand, we also examined optimizations that essentially trade additional computation and communication costs for memory, or additional computation for memory and communication balance. Filtering-out singleton *k-mers* via a Bloom filter saves large amounts of memory and minimized the amount of wasted computation on singletons (Section 3.4). Then again, it also costs as much computation and communication as the hash table (graph) construction itself (Figure 3.18). While detecting and specially handling skew from heavy hitter *k-mers* greatly improved subsequent *k-mer* exchanges, we showed the cost of the optimization does not always outweigh the benefit (Section 3.7). Similarly, we presented cost-benefit trade-offs of precise cardinality estimation with HyperLogLog in Section 3.6. We showed that while it is possible to get highly accurate $k$-mer set cardinality estimates, and doing so does minimize the amount of memory used by the Bloom filter, the absolute saving thereof must be weighed against the cost of streaming the bag of *k-mers*. We highlighted that simple overestimates based on the *k-mer* bag size (see also Section 3.3) can suffice for the purposes of both applications.

This case study offers several takeaways for irregular application parallelization in general. It demonstrates that simple complexity analysis is not a tool we should underestimate in the face of complicated irregular application parallelization. Though there are many sources of irregularity in this application's data, e.g. read lengths and distributions, error rates, and so on, by identifying the dominant term in the complexity analysis, we have identified the source of irregularity requiring the most attention (the *k-mers*). The analysis relating *k-mer* lengths to error rates, read lengths, coverage depth and genome size confirms that all these other sources of irregularity are managed by managing the irregularity from *k-mers*. Furthermore, extending the analysis with the $\alpha - \beta$ model, we have approximated the scales at which it pays-off to employ optimizations that trade more computation for improved communication performance i.e. the computation-communication cross-over for any given workload. We also highlighted the memory dimension to design decisions with respect to communication (dictating communication batch sizes and iterations) and computation (determining whether or when the Bloom filter optimization is necessary).

In general, memory management is often considered an orthogonal consideration in analytical performance modeling, especially in distributed systems in which additional memory is "free" (but unfortunately only in theory). Though aggregate memory may be unlimited with the addition of more nodes, in practice, the ratio of computation to memory resources on-node is limited, and maximally utilizing computational resources on-node may not be possible with the available on-node memory. Further, application users have limited node resources, and may not be able to run at the scales required by codes that always, for example, trade memory to lessen computation.

Thus we have highlighted in this chapter the interdependencies of memory, computation,

and communication design for our irregular applications. This is a theme we shall revisit throughout the thesis, ultimately presenting a case for all three to be included in performance model for irregular applications. Such models are relevant to application designers and hardware architects making decisions regarding balancing memory, computation, and communication consumption and resources. As an inseparable consideration, we will also consider an alternative programming paradigm for balancing these constraints in irregular applications. Before we do so however, the next chapter considers a bulk-synchronous approach to long read overlap hypergraph traversal and refinement, building on the bulk-synchronous overlap hypergraph construction presented in this chapter.

# Chapter 4

# Cross-Architectural Analysis of Bulk-Synchronous Overlap Hypergraph Construction, Traversal, & Refinement

This chapter builds on the long read overlap hypergraph construction, designed for distributed bulk-synchronous parallelism, presented in the previous chapter. Given the overlap hypergraph, built using short common subsequences (*k-mers*), hypergraph traversal and refinement is necessary to verify and to determine the extent of overlap between reads sharing *k-mers*. As described in Chapter 2, hyperedges in the hypergraph are reads sharing multiple intersecting sets of *k-mers*. Hyperedge set intersections indicate potential overlap between reads. To determine whether and how reads truly overlap, given that the reads contain errors at high rates, read-to-read alignments are computed. The alignment information for reads that truly overlap is essential for direct analysis of the underlying genome, for long read error correction, and for assembly. Irregularity in the overlap hypergraph and the read lengths alone present challenges for balanced distributed memory computation and communication.

This chapter presents a bulk-synchronous approach to overlap hypergraph traversal and refinement in DiBELLA. DiBELLA's hypergraph construction presented in the previous chapter, with the traversal and refinement presented in this chapter, constitute the first long read to long read alignment pipeline designed for distributed memory scalability.[1] However, the larger focus of this chapter is the balance of memory, computational, and communication capacities that support efficient bulk-synchronous scalability of this application. This will be explored through extensive empirical performance analysis across three HPC architectures and one cloud offering. Each represents different balancing points between communication, memory, and computational resources. Though the hypergraph construction step is not re-

[1]Some of this work was originally published in "DiBELLA: Distributed Long Read to Long Read Alignment" by M. Ellis et al. [13].

described in this chapter (see the previous chapter, Chapter 3), its performance is included in the performance analysis of the pipeline as a whole.

## 4.1 Bulk-Synchronous Parallel Pipeline Overview

DiBELLA [13] is a 3-stage software pipeline for distributed memory scalable many-to-many long read overlap and alignment. We refer to the 3 stages as overlap hypergraph construction, traversal, and refinement. DiBELLA's Single Program Multiple Data (SPMD) bulk-synchronous pipeline is illustrated in Figure 4.1. It begins by partitioning the reads by size in memory over the processors, labeled $P_0, P_1, ..., P_{n-1}$ in Figure 4.1, via parallel I/O. As described more thoroughly in Chapter 3, batches of *k-mers* are computed in parallel from read partitions, filtered, and re-balanced via one or more many-to-many *k-mer* exchanges. A complete global *k-mer* histogram is optionally saved for the user, and can be collected by running the whole pipeline or just the first stage in standalone mode.

The *k-mers*, filtered by global frequency, are stored in local partitions of a distributed hash table. Along with the *retained k-mers* and their counts, *k-mer* metadata, specifically the locations of instances of the respective *k-mer* in the input read set, are stored in hash table value tuples (see Figures 3.3-3.4). The hash table partitions are traversed independently in parallel to compute candidate overlaps, which correspond to pairwise alignment tasks for the next stage (see Figure 4.3). Due to the *k-mer* load balancing strategy described in Chapter 3, the owner of any given retained *k-mer* is not guaranteed to also be the owner of any or all of the reads in which the *k-mer* appears. Hence, candidate overlaps, represented by pairs of read identifiers, are consolidated at the owner of one or both of the indicated reads via a many-to-many exchange. The locations of the *retained k-mer* instance in the respective read pair are communicated with the read identifiers. The *k-mers* common to any read pair can be recomputed locally, but the *k-mers* stored in the hash table were filtered by (global) characteristics of the data. The filtered *k-mers* are used to seed respective pairwise seed-and-extend alignments in the third stage. The second stage is described more fully in Section 4.3. The first two stages can be run standalone to collect candidate overlaps without alignment information – the unrefined hypergraph.

The third and final stage computes all pairwise alignments for all candidate overlaps identified in the previous stage. Before the computation, a many-to-many exchange of the reads necessary to more than one processor (cut hyperedges in the hypergraph partitioning, see Chapter 2, Section 2.3) is performed. Once these reads are exchanged, the many-to-many alignment computation can proceed independently in parallel. The refined hypergraph with pairwise alignment information is output via parallel I/O. This stage is described more fully in Section 4.4. This pipeline was initially implemented in MPI 2.0, using MPI Alltoall and Alltoallv for the many-to-many exchanges.

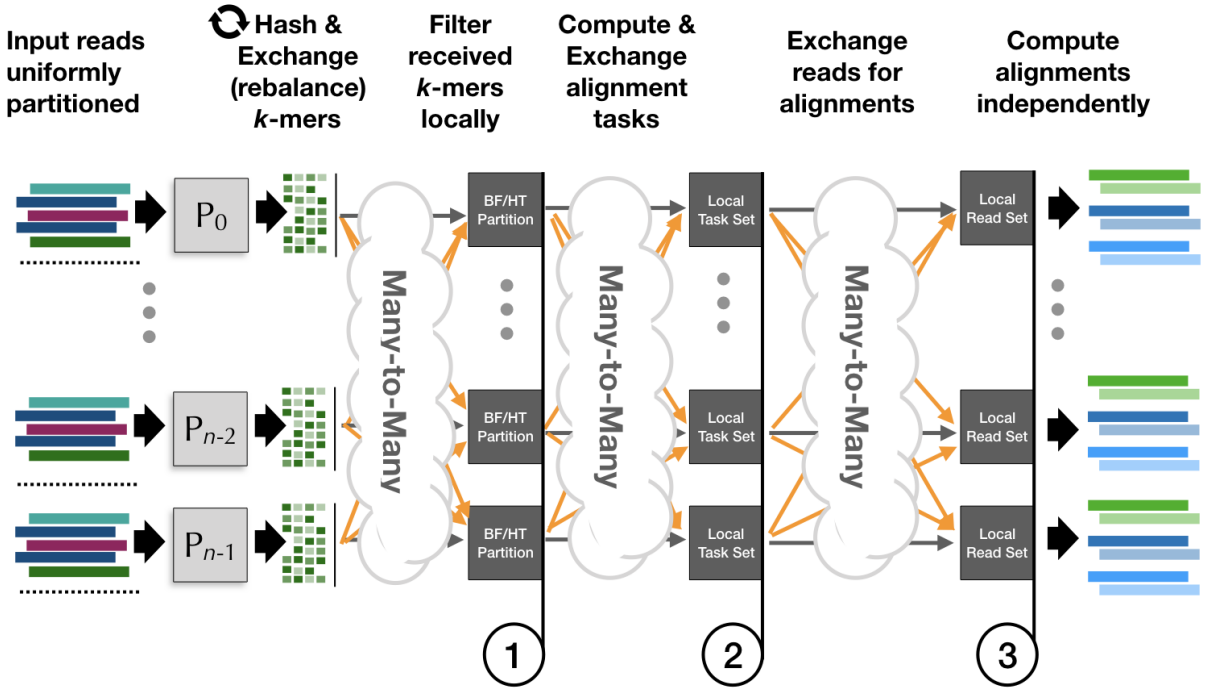


Figure 4.1: An illustration of DiBELLA's 3-stage Single Program Multiple Data (SPMD) bulk-synchronous pipeline. By the end of step 1, the initial overlap hypergraph is constructed. By the end of step 2, hyperedge set intersections, identifying alignment tasks for the last step, are computed and re-balanced. By the end of step 3, the hypergraph has been refined, and pairwise alignment information for overlapping reads is ready for output.

## 4.2 Empirical Analysis Overview

Since empirical results are included in each of the following sections, alongside the respective algorithm descriptions, we begin with an overview of the empirical evaluation procedures. Our experiments were conducted on four computing platforms, which include HPC systems with varying balance points between communication and computation, as well as a commodity AWS cluster. This gives us performance insights into trade-offs between extremes of network capabilities. Evaluated platforms include the Cori Cray XC40 and Edison Cray XC30 supercomputers at NERSC, the Cray XK7 MPP at the Oak Ridge National Lab, and an Amazon Web Services (AWS) c3.8xlarge cluster. Details about each architecture are presented in Table 4.1. Titan has GPUs and CPUs on each node, but we use only the CPUs and total 16 *Integer* [2] cores per node. AWS does not reveal specifics about the underly-

[2]There are technically 8 "full" cores per node since the microarchitecture splits certain resources; we adopt the common albeit somewhat imprecise description, "Integer" cores, throughout to disambiguate. In all experiments presented, MPI ranks are mapped to exclusive L1 caches and Integer Units since these are most important to the application.

Table 4.1: Evaluated platforms. *128 byte Get message latency in microseconds. †Using the optimal number of cores per node. ‡Measured over approx. 2K cores. §MB/s with 8K message sizes. $^{\alpha}$CPUs only.

| Processor | Cori I Cray XC40<br>Intel Xeon (Haswell) | Edison Cray XC30<br>Intel Xeon (Ivy Bridge) | Titan Cray XK7$^{\alpha}$<br>AMD Opteron |
|---|---|---|---|
| Freq (GHz) | 2.3 | 2.4 | 2.2 |
| Cores/Node | 32 | 24 | 16 |
| Intranode LAT*† | 2.7 | 0.8 | 1.1 |
| BW/Node†‡§ | 113.0 | 436.2 | 99.2 |
| Memory (GB) | 128 | 64 | 32 |
| Interconnect | Aries Dragonfly | Aries Dragonfly | Gemini 3D Torus |

ing node architecture or interconnect topology, other than an expected 10 Gigabit injection bandwidth. Based on our measurements, the AWS node has similar performance to a Titan CPU node. Both data sets are small enough to fit in the memory of a single node, and in all experiments, MPI Ranks are pinned to cores.

Figure 4.2 shows the pairwise alignment computation almost entirely dominating the runtime with near linear speedups. We currently employ an even more recent version of an efficient CPU implementation of the X-drop seed-and-extend algorithm[69] from the SeqAn library[10]. Given that algorithmic and software improvements to (pairwise) alignment constitute an old yet highly active research area, we expect to and have already benefitted from improvements in pairwise alignment software. In order to reveal cross-network performance and communication bottlenecks while pairwise aligners continue to improve, most of our experiments use an input data set and runtime parameters that result in minimal computational intensity.

The primary data set in our experiments is from the *E. coli* bacterial genome, sequenced with a depth of $d = 30$ (referred to hereforward as *E. coli* 30×). It consists of $16,890$ long reads from the from *Escherichia coli* MG1655 strain, resulting in a 266 MB input file. It was sequenced using PacBio RS II P5-C3 technology and has an average read length of $9,958$ base pairs (bps). The second data set, *E. coli* 100×, was sequenced using PacBio RS II P4-C2 with a depth of $d = 100$. It consists of $91,394$ long reads from the same strain, with an average read length of $6,934$ bps, resulting in a 929 MB input file. DiBELLA's hypergraph construction and traversal identifies approximately 2.3 million potentially overlapping read pairs for the first data set and 24.9 million for the second one.

Computational intensity is most affected by the number of alignments performed for each pair of reads, since each pair shares varying numbers of *k-mers* (alignment seeds). Some of these seeds reflect a shifted version the same overlapping region, whereas others may be independent (and ultimately incorrect) overlaps. We use three different workload configurations to vary computational intensity. At the two extremes, the *one-seed* configuration computes a seed-and-extend pairwise alignment on exactly one seed per pair, while the *all-*

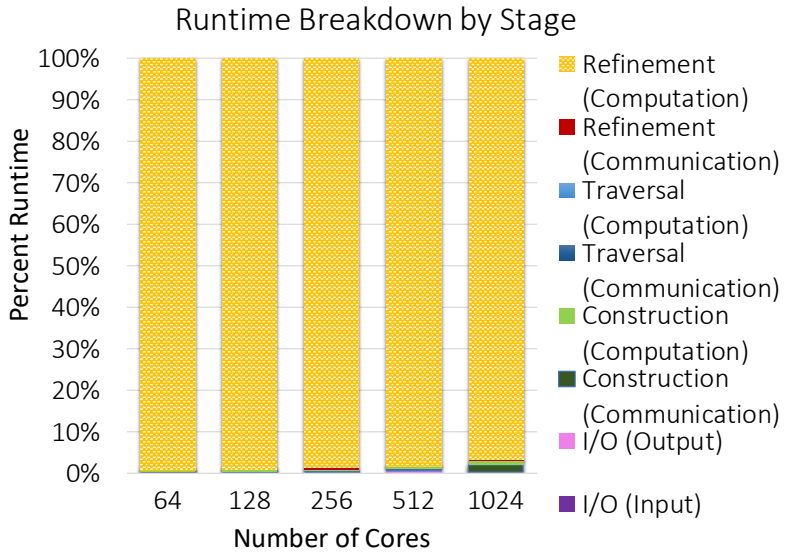


Figure 4.2: Strong scaling (percent) runtime breakdown of the DiBELLA pipeline from 4 to 64 nodes on Titan, each running 1 MPI rank per *Integer* core / L1 cache (16 total per node). All pairwise seed-and-extend alignments for all seeds (*k-mers*) discovered and retained from hypergraph construction were computed. The time spent in any component (communication, computation, or I/O) of any other stage (construction, traversal, refinement) are barely visible even at 64 nodes / 1,024 cores because the pairwise alignment computation (yellow) in the refinement stage almost completely dominates the runtime (from 99.4% to 96.8% across scales). Over 4 nodes / 64 cores, the pairwise alignment computation speedup is 10.1× (versus a perfect 16×), and the overall speedup is 9.9×. At 1,024 cores, the second bottleneck is the construction stage, though it is still less than 3% of the runtime. Overall, the runtime was reduced from 1.8 hours to 10 minutes.

*seeds* configuration computes a seed-and-extend pairwise alignment on all the available seeds separated by at least the *k-mer* length, which is $k = 17$ for these workloads. We denote the minimum distance between seeds as $q$; e.g. for the *all-seeds* workload, $q = k = 17$. As an intermediate point we consider only seeds separated by at least $q = 1$Kbps - a setting used in quality-focused analyses in related work [23]. In terms of computational intensity, this is an intermediate point between the *one-seed* and *all-seeds* extremes because, while it produces a filtered subset of all seeds, we expect more than one seed per read pair given typical read lengths in range $[10^3, 10^5]$. For the *E. coli* 100× data set, the average number of seeds with $q = 1$Kbps is 1.8 per candidate overlap; the workload computes approximately 45.7 million seed-and-extend alignments total. Similarly, for the *E. coli* 30× workload with $q = 1$Kbps, the average number of seeds per candidate overlap is 2.2; approximately 5.1 million seed-and-extend alignments are computed overall. For all workloads, a fixed, relatively low x-drop value is used, so that attempts to align reads that do not overlap fail fast

- avoiding excessive computation. All data sets are sufficiently small that the working set size fits on a single node across the platforms in our comparison. This choice enables us to show the performance impact of intra-node to inter-node communication on the overall pipeline performance and highlight scaling bottlenecks, and to explore strong scaling on a modest number of nodes, important for comparison with AWS. Memory requirements and the relationship thereof to parallelization approaches is a topic of the next chapter.

## 4.3 Hypergraph Traversal & Task Redistribution

Chapter 3 describes the process of building the initial overlap hypergraph, represented with a distributed hash table that stores both the *k-mer* frequency and the locations of respective *k-mers* in the original reads. The distributed hash table maps retained *k-mers* to their source locations (sets of read identifiers, "RIDs", and corresponding positions). Abstractly, reads are hyperedges sharing multiple *k-mer* vertices. Hyperedge intersections are candidate read overlaps. To compute the intersections, we traverse the hypergraph, identifying overlapping read pairs, and consolidate the *k-mers* shared between every read pair. These *k-mers* will be used for pairwise seed-and-extend alignment in the refinement step. We hereforward refer to these read pairings with their *k-mers* as computational or alignment "tasks". The distributed hash table is a compressed vertex-centric representation of this sparse hypergraph. Once it is constructed, traversing the hypergraph to compute hyperedge set intersections (identify read pairs that share *k-mers*) is straightforward.

The traversal is performed independently in parallel from hash table partitions. Algorithm 1 illustrates this simple, direct computation of the set of all pairs of reads represented by identifiers $(r_a, r_b)$, where $r_a$ and $r_b$ share retained *k-mer* (s). Each *k-mer* "contributes" to the discovery of $[2, m(m-1)/2]$ read pairs where $m$ is the maximum frequency retained *k-mers*. This frequency, $m$, is calculated according to the BELLA model [23] or defined by the user. Each of these represents a candidate overlap, which is verified or pruned in the refinement step via pairwise alignment. However, according to the uniformly random distribution of *k-mers* and the independent partitioning of the reads for load balanced hypergraph construction (see Chapter 3), the owner of the *k-mer* matching $(r_a, r_b)$ may not be the owner of either involved read. To maximize locality in the alignment stage (minimize the movement of reads) each task is buffered for the owner of $r_a$ or $r_b$ (which may be the same owner), according to the simple odd-even heuristic in Algorithm 1. Reads in the input are unordered and partitioned uniformly by size in memory. Hash table values (RID sets) are also unordered. Hence, for fairly uniform distributions of reliable *k-mers* in the input, we expect this heuristic to roughly balance the number of alignment tasks assigned to each processor. Under the assumption that the input file is not externally reordered between runs, the load balancing strategy is deterministic across runs of the same scale. Load balancing by (number) of tasks is still imperfect, since individual pairwise alignment tasks may have different costs in the alignment stage. The computational impact of various features, such as read lengths and *k-mer* similarity, could be used for estimating the cost changes within the

pairwise alignment kernel. We leave further analysis of the relationship between the choice of pairwise alignment kernel and overall load balancing to future work. Our expectations of the general load balancing strategy are discussed further with empirical results in the context of the alignment stage description, Section 4.4. The final steps of the traversal are the irregular all-to-all (or many-to-many) communication of buffered tasks, implemented with MPI_Alltoallv, and the (optional) output of the candidate overlaps.

**Algorithm 1:** Parallel (SPMD) hash table traversal

**Result:** All pairs of reads sharing at least 1 retained *k-mer* in hash table partition, H, and corresponding *k-mer* positions (elided) are composed into alignment tasks. Each task, with read identifiers $(r_a, r_b)$, is stored in a message buffer for the owner of $r_a$ or $r_b$. Let $m_{kmer}$, such that $m_{kmer}, \leq m$ be the number of reads associated with a given *k-mer*, identified by the key $k_{hash}$.

```
for each k_hash in hash table H do
    for i = 0 to (m_kmer − 2) do
        for j=i+1 to (m_kmer − 1) do
            (r_a, r_b) = task(H[k_hash][i] , H[k_hash][j],...)
            if (r_a%2 = 0 AND r_a > r_b) OR (r_a%2 ≠ 0 AND r_a < r_b) then
                buffer[owner(r_a)]← (r_a, r_b)
            else
                buffer[owner(r_b)]← (r_a, r_b)
            end
        end
    end
end
```

Neither the number of overlapping read pairs nor the number of retained *k-mers* common to each pair can be determined for a given workload until runtime. However, from a few simple observations, we can derive a practical estimate of the number of retained *k-mers* before processing the entire input. As detailed in Chapter 3, the total number of *k-mers* parsed from the input is roughly equal to the number of characters in the input (Formula 3.2), and the size of this *k-mer* multiset in memory is one order of magnitude larger than the input, given typical values of $k \in [14, 17]$. However, only distinct *k-mers* with frequency within [2, $m$] in the *k-mer* multiset are retained and stored, where $m$ is the maximum frequency of retained *k-mers*. As in Chapter 3, let $K_{bag}$ be the *k-mer* multiset ("bag" for short), and its cardinality be $|K_{bag}|$. To distinguish the *k-mer* multiset from the set of all *k-mers*, let $K_{set}$ be the set of all *k-mers*, and $|K_{set}|$ be its cardinality. Likewise, let the subset of retained *k-mers* of interest to the application be $K_{subset}$, and $|K_{subset}|$ be its cardinality. Note, we expect $K_{bag}$ to be strictly larger than the corresponding set of all *k-mers*, $K_{set}$, and both to be larger than the subset of retained (non-singleton) *k-mers*, $K_{subset}$. Let the ratio of retained *k-mers* to the total number of *k-mers* be $\iota_{bag} = |K_{subset}|/|K_{bag}|$, and the ratio of retained *k-mers*

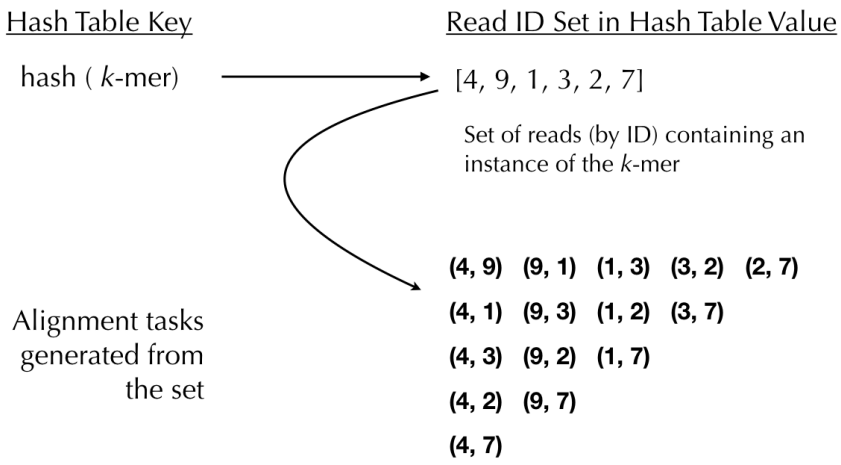


Figure 4.3: Example: computing candidate overlaps from a hash table element. Reads are represented by integer identifiers in the hash table value tuple; *k-mer* locations and frequencies are elided from the value tuple for simplicity. See Figures 3.3-3.4 for full illustrations of hash table elements.

to the *k-mer* set size be $\iota_{set} = |K_{subset}|/|K_{set}|$. The importance of the distinction is that $|K_{subset}|$ cannot be known until the *k-mer* set is computed and filtered, which in our case, is done during the hypergraph construction step described in Chapter 3. $|K_{bag}|$, on the other hand, can be very closely estimated from the number of characters in the input (Chapter 3, Section 3.3, Formulas 3.1-3.2). Further, $|K_{set}| \leq |K_{bag}|$, and thus $\iota_{set} \geq \iota_{bag}$. We can think of $\iota_{bag}$ and $\iota_{set}$ as general "filtering factors". In our cross-genome analysis, $\iota_{set} \in [0.04, 0.12]$. In practice, observed values of $\iota_{bag}$ and $\iota_{set}$ can be used to roughly estimate the cardinality of the retained *k-mer* subset, $|K_{subset}|$, before processing the entire input at runtime. This is useful in the absence of a model for the frequency of unique or rare *k-mers* in genome data sets.

The analysis overall is useful for estimating the overlap computation and communication costs as well, and applicable beyond our particular implementation. An upper bound on the total (global) number of overlaps follows in Equation 4.1. The lower bound (Equation 4.2) follows from the fact that retained *k-mers* must occur in at least two distinct reads (identifying at least one overlap) or they are discarded. The parallel computational complexity of Algorithm 1 (with $P$ parallel processors) is shown in Formula 4.3, which assumes constant-time storage of read pair identifiers. The hidden constant in Formula 4.3 is halved by exploiting asymmetry.

$$O(\iota_{set} \cdot K_{set} \cdot m^2) < O(\iota_{bag} \cdot K_{bag} \cdot m^2) \tag{4.1}$$

$$O(\iota_{set} \cdot K_{set}) < O(\iota_{bag} \cdot K_{bag}) \tag{4.2}$$

$$O(\frac{\iota_{set} \cdot K_{set} \cdot m^2}{P}) \tag{4.3}$$

Ignoring the constant for the memory size of a pair of read identifiers and positions, the aggregate communication volume is also bounded above by Equation 4.1, and below by Equation 4.2.

As a last computational step, after tasks are computed from the parallel traversal and communicated, the consolidated lists of common *k-mers* may be filtered further depending on certain runtime parameters. That is, some subset of the set of *k-mers* shared by potentially overlapping read pairs will be used to seed the alignments in the refinement step. This subset may be equal to the whole set, or a proper subset determined by certain runtime parameters. These runtime parameters can be thought of as "exploration" constraints. They include the minimum required distance between seeds, denoted $q$, and the maximum number of seeds to explore per candidate-overlap. In general, increasing $q$ and/or decreasing the maximum number of seeds for pairwise alignments can decrease the computational intensity. Decreasing $q$ and increasing the maximum number of seeds can increase the computational intensity. The seed positions and numbers are ultimately determined by the workload, however, so shifting these parameters does not guarantee a shift in computational intensity. Note also, a discussion of these settings in relation to overlap verification accuracy versus computational cost is presented in the BELLA analysis [23]. In general, increasing the number of seeds to explore per overlap increases computational cost of the alignment stage; but not necessarily linearly, as it depends on the pairwise alignment kernel employed and its parameters. Because the number of seeded alignments to try for a given pair of reads is still an open research topic, we present results varying the number of seeds in Section 4.4.

Strong scaling results for the hypergraph traversal are shown in Figure 4.4. These are presented across our evaluated platforms in terms of millions of retained *k-mers* processed per second. One unexpected feature of this hypergraph is the dip in Cori's performance trend at 16 nodes, due to an unexpected spike in the communication exchange time that does not continue to 32 nodes. This result was consistent across repeated experiments, varying 32 node allocations, and even considering the minimum exchange times as well as the average and median exchange times across all runs. We suspect it is the result of a performance bug in the MPI_Alltoallv implementation for this particular configuration, communication volume, and data layout. Because the traversal and exchange is dominated by this communication time at 16 nodes, the spike brings the Cori performance down to Titan and AWS's at 16 nodes.

Additionally, it is notable that the AWS cluster slightly outperforms the Titan supercomputer from 1 to 8 nodes. Only from 16 to 32 nodes does Titan finally overtake AWS. A broken down view of their respective strong scaling efficiencies, in terms of *Exchange*, *Computation*, and *Overall* efficiency, is shown in Figures 4.5-4.6. Efficiency is calculated relative to performance on 1 node, employing all 16 cores; an efficiency of 1 is perfect. In figure 4.5, the computation on AWS speeds-up super-linearly up to 3.1× at 32 nodes, due to shrinking partition sizes and good cache behavior. The AWS *Exchange* efficiency, however, drops from 1 to just 0.33 between 1 to 2 nodes, and continues to degrade near exponen-

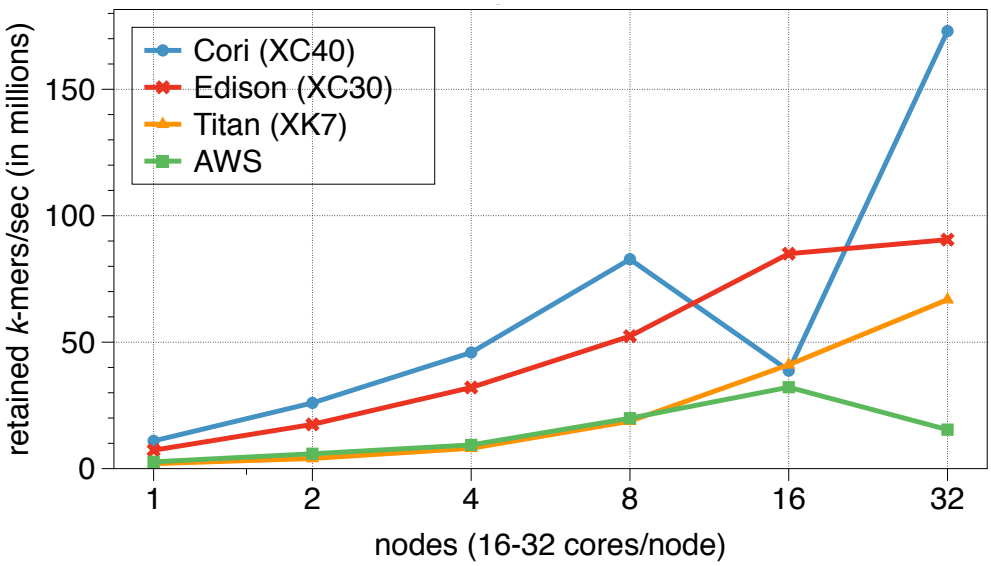


Figure 4.4: Cross-architecture performance of the traversal and task exchange in millions of retained *k-mers*/second given *E. coli* 30× one-seed.

tially. Trend lines fitted to the computation and exchange efficiency are shown in the same figure; the exponents are 0.2 and −0.9 for the computation and exchange, respectively. The *Overall* efficiency is determined primarily by the efficiencies of both the *Computation* and the *Exchange*, and so hovers around 1 between 1 to 8 nodes, drops to 0.75 at 16 nodes, and finally to 0.17 at 32 nodes as the workload becomes increasingly communication-dominated. Titan's strong scaling efficiency is shown in Figure 4.6. The computation speeds-up super-lineary but not quite as quickly as the computation on AWS; the fitted trend line shown alongside the results is roughly $y = x^2$. From absolute runtimes for the same workloads, it appears AWS has the faster cores with potentially larger caches than Titan's *Integer* cores. Titan's communication efficiency degrades as we strong scale, but also not quite as rapidly as the AWS communication efficiency; the fitted trend line is roughly $y = x^{-1/2}$. The result is that Titan maintains efficiency close to 1 as we scale out, and even slightly exceeds 1 at 16 nodes. While both workloads are communication-dominated between 16 to 32 nodes, the performance of Titan's HPC interconnect ultimately results in Titan overtaking AWS in the overall performance comparison shown in Figure 4.4.

## 4.4 Overlap Hypergraph Refinement via Many-to-Many Alignment

The *k-mer* load balancing strategy described in Chapter 3 enables uniform *k-mer* load balancing in the overlap hypergraph construction, and in its traversal described in Section 4.3. Every pair of reads sharing one or more *k-mers* is a candidate-overlap. A pairwise alignment

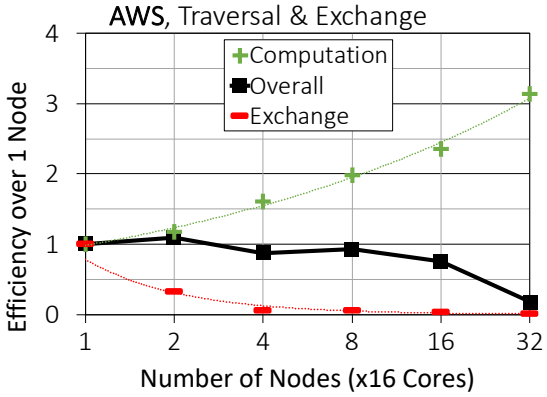


Figure 4.5: Strong scaling efficiency of the traversal and exchange on AWS with the *E. coli* 30× one-seed workload. Includes fitted trendlines of roughly $y \approx e^{0.2x}$ for the computation and $y \approx e^{-0.9x}$ for the exchange.

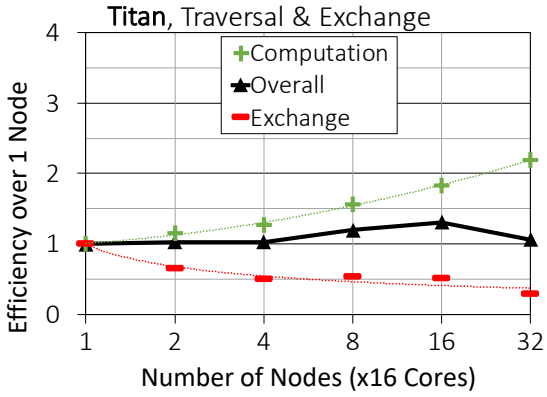


Figure 4.6: Strong scaling efficiency of the traversal and exchange on Titan with the *E. coli* 30× one-seed workload. Includes fitted trendlines of $y \approx x^2$ for the computation and $y \approx -\sqrt{x}$ for the exchange.

is computed for every candidate-overlap to see if the alignment score meets some minimal threshold, indicating that the two reads, in all likelihood, came from intersecting regions of the sequenced genome. Candidate-overlaps that do not meet the scoring criteria are pruned from the hypergraph. The alignment information for retained vertices is included in the final output, as it is essential for any direct analysis, error correction, and assembly of long reads. We refer to the process of traversing and computing pairwise alignments for vertices in the hypergraph as *overlap hypergraph refinement* (also described in Chapter 2).

The hypergraph vertices, initially, are all candidate-overlaps, also called alignment tasks. The reads required by multiple candidate-overlaps or alignment tasks are the hyperedges. Any given mapping of the vertices to parallel processors may cut hyperedges. Hence the data represented by hyperedges (reads) will need to be communicated across the respective cuts eventually. Only *k-mers* and read identifiers (not the actual reads) are communicated in earlier steps of the pipeline. Note, though all *k-mers* shared between any two reads could be recomputed, we instead store these specific *k-mers* because these are the locations of (globally) rare *k-mers*, and information on the global prevalence of *k-mers* cannot be recomputed from just the pair of reads. After the traversal and subsequent task exchange, each alignment task with its list of shared *k-mer* positions, is stored with the owner of at least one, but not necessarily both, of the involved reads. Computing the pairwise alignment of any candidate-overlap necessarily involves both reads.

The properties of the overlap hypergraph underpin the initial communication design for our application. The size of the retained *k-mer* set determines the size and sparsity of the overlap hypergraph. From our filtering steps, we expect this hypergraph to be sparse; from empirical observations across data sets, the filtering typically reduces the *k-mer* set size

by 85-98%. To effectively maximize locality and bandwidth utilization under these conditions, we first explore the performance of a bulk synchronous exchange implemented via MPI_Alltoallv. Note that once the reads are communicated, the alignment computation can proceed independently in parallel. We expect that speedups from the subsequently embarrassingly parallel alignment computations (which are quadratic for exact pairwise alignment and at least linear in the length of the long reads for approximate pairwise alignment) will compensate for inefficiencies in the communication to some workload-dependent degree of parallelism.

Each candidate-overlap shares an unknown *a priori* number of retained *k-mers* that are used to seed the alignment. Any number between one or all of these seeds will be explored in application runs, depending on the user's objectives and runtime settings. For the purposes of genome assembly, for example, the alignment should find only one single best alignment between any read pair (covering the region of the genome from whence they originated during sequencing), or the alignment computation should determine that the read pair does not actually overlap despite sharing some short *k-mer* matches. Figure 4.7 shows performance (alignments per second) across our evaluated platforms using one seed per alignment, the setting with minimum computational intensity. Here, the number and speed of the cores per node determine the relative performance ranking (see Table 4.1), with Cori's 32 cores/node clearly surpassing the other systems.

The load balancing strategy described in Sections 4.3-4.4 produces near perfect load balancing in terms of the number of alignments computed per parallel process, but imperfect load balancing in terms of time to exchange and compute all alignments. Figure 4.8 shows the latter load imbalance, calculated as maximum per rank alignment stage times over average times across ranks (1.0 is perfect). There are two reasons for this load imbalance in terms of compute and exchange costs: (1) reads have different lengths, which effect both the exchange time and the pairwise alignment time, (2) the X-drop algorithm returns much faster when the two sequences are divergent and does not compute the same number of cell updates. A smarter read-to-processor assignment could optimize for variable read lengths, eliminating the exchange imbalance. However, the imbalance due to x-drop can not be optimized statically as it is not known before the alignment is performed. To mitigate the impact of (2), one would need dynamic load balancing, which is known to be high-overhead in distributed memory architectures. The load imbalance in terms of the number of alignments performed per processor is less than 0.002% across all machines and scales. Future work should consider not only the number of alignments per processor but other kernel-dependent characteristics affecting the cost of each pairwise alignment.

## 4.5 Overall Pipeline Performance

The performance rates of each stage show similar results across machines, with the more powerful Haswell CPU nodes and network on Cori (XC40) giving superior overall performance. As expected, all-to-all style communication for irregular many-to-many exchanges

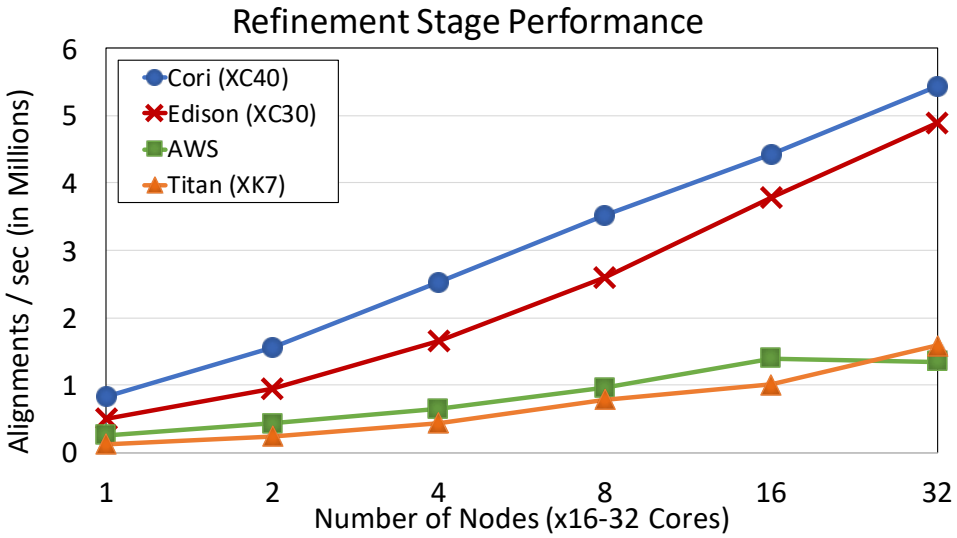


Figure 4.7: Cross-architecture hypergraph refinement stage performance in millions of alignments / second, strong scaling the *E. coli* 30× one-seed workload.

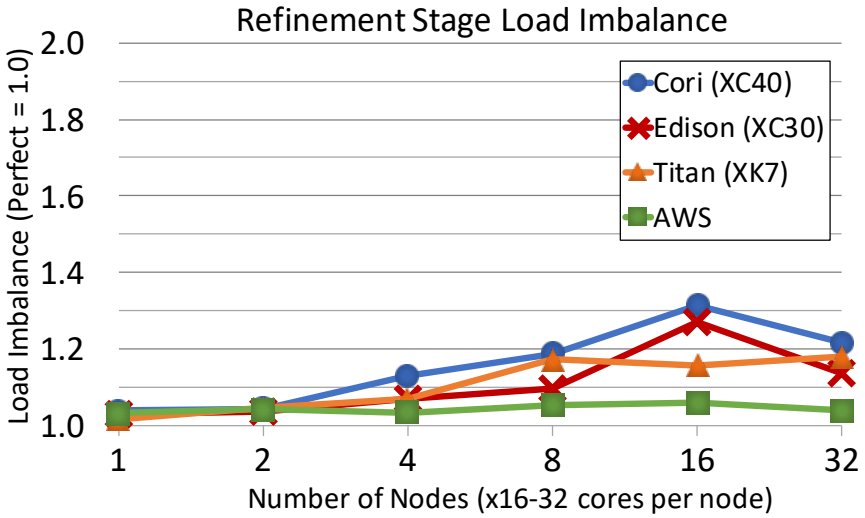


Figure 4.8: Load imbalance of the hypergraph refinement stage, strong scaling *E. coli* 30× one-seed. Load imbalance is calculated using maximum over average stage times across ranks (1.0 is perfect). The apparent spike at 16 nodes on Cori and Edison is due to the fact that communication time is included in this metric, and that there is a spike in communication time on Edison and Cori at 16 nodes as discussed in Section 4.3.

scales poorly on all networks, but especially with the commodity AWS network. Somewhat more surprising is the high level of superlinear speedup on some stages once the data fits in cache or other memory hierarchy level. The question for overall performance is how these two effects trade off against one another and how the stages balance out. Figure 4.9 shows diBELLA's overall pipeline efficiency on Cori, across varying workloads and computational intensity. Two data sets are used, *E. coli* 30x and *E. coli* 100x, and 3 seed constraints, one-seed, all seeds separated by at least $q = 1$Kbps, and all seeds separated by $q = k = 17$ bps, in Figure 4.9. Clearly, increasing the computational intensity with larger inputs and seed counts does not alone determine overall efficiency. While the computational efficiency increases with higher computational intensity, the overall efficiency is significantly impacted by the degrading efficiency of exchanges.

Let us consider runtime breakdowns by stage. Note, the pipeline is implemented in the logical stages of construction, traversal, and refinement, so that each (primarily the dominant stages of construction and refinement) can be run standalone as needed. This design facilitates our performance analysis with respect to runtime breakdowns by stage as well. Figures 4.10 and 4.11 respectively show runtime breakdown by stages on Cori for *E. coli* 30× exploring 1 seed per overlapping pair of reads, and *E. coli* 100×, using all seeds separated by a minimum of $q = 1$Kbps. As described in Section 4.2, the *E. coli* 30× one-seed workload is a minimally compute-intensive workload in our study. Relative to this workload, the *E. coli* 100× input file is 3.5× larger, and the configuration with $q = 1$Kbps computes roughly 20× more alignments. In Figures 4.10 and 4.11, the communication time is broken out for each pipeline stage. The stages are fairly evenly balanced, although alignment is more computationally expensive than the other stages (and dominates to 32 nodes in the more computationally intense workload). Focusing on Figure 4.10, the communication time in the Bloom filter stage is surprisingly higher than the rest of the construction stage. The rest of the construction stage initializes the distributed hash table; its communication volume is 2.5x higher and its communication pattern and number of messages is identical. Further investigation revealed that the problem is the first call to the MPI Alltoallv routine, which is almost twice as expensive the first time as the second, so later calls benefit from whatever internal data structure and communication initialization happened in the Bloom Filter step. This effect was visible to varying degrees on all 4 platforms. This kind of behavior is most noticeable for workloads with lowest computational intensity.

To further drill down on network and processor balance, Figure 4.12 shows the overall efficiency of the pipeline, along with the exchange efficiency, across all 4 networks. From an efficiency standpoint, the Cray XK7 using only the CPU features on each node gives the best network balance for this problem, even though the network is an older generation than on the XC30 and XC40.

Let us now consider performance in terms of alignments computed per second (dubbed "alignment rate") over the whole pipeline run, for which the total number of alignments is fixed for a given input configuration. Results for each architecture are shown in Figure 4.13. While not well balanced for efficiency, the higher speed processor and network on Cori (XC40), outperform the others in running the full diBELLA pipeline. The performance

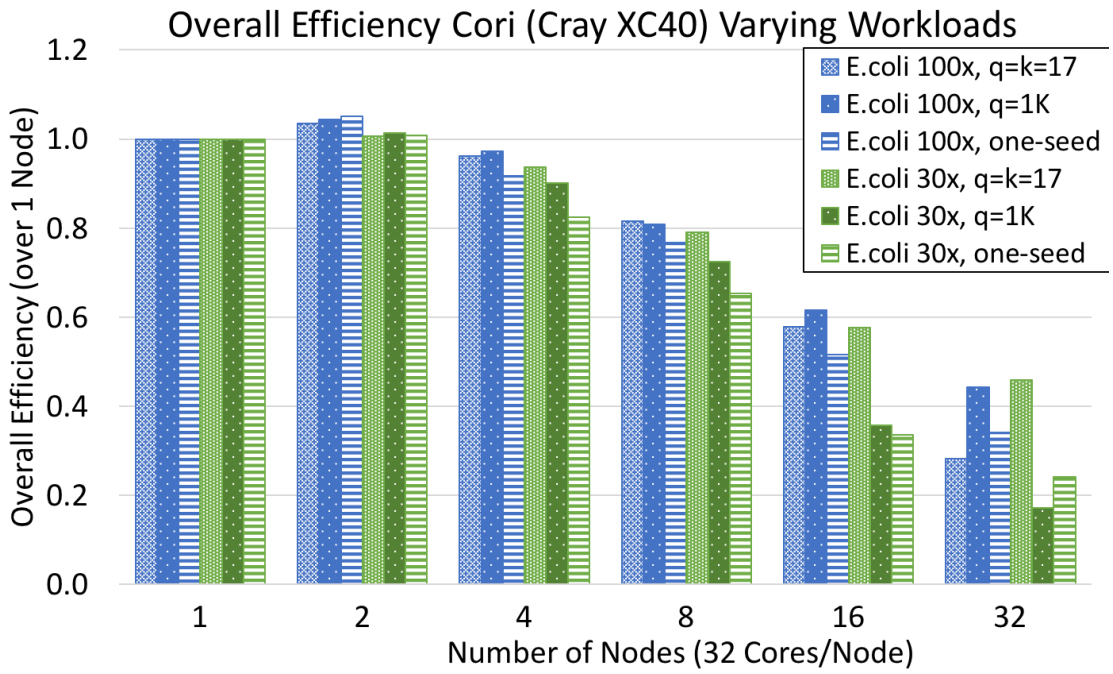


Figure 4.9: Overall strong scaling efficiency, relative to one node, on Cray XC40 (Cori) over 2 data sets, *E. coli* 30× and *E. coli* 100×, varying seed constraints (one-seed, all seeds separated by $q$ =1K characters, and all seeds with $q = k = 17$) for each data set.

anomaly at 16 nodes for Cori, which was seen in the other stages, is apparent here in the overall performance, probably due to a performance issue in the MPI implementation. With that exception and an apparent drop on AWS at 32 nodes, due to the relative network performance, all of the systems show increasing alignment rates with increasing node counts. Recall, that our standard problem used here (*E. coli* at 30x coverage with only 1 seed used per read pair for alignment) was specifically chosen as the low end of computational intensity, and so highlights scaling limits of the machines.

## 4.6 Summary & Conclusions

We presented a bulk-synchronous approach, for irregular hypergraph construction, traversal, and refinement for long read overlap hypergraphs. Our implemention, DiBELLA, is the first distributed memory scalable software for this problem, making it possible to analyze data sets that are too large for a single shared memory and or making heroic computations routine. We performed a thorough performance analysis on three leading HPC platforms as well as one commodity cloud offering, showing good parallel performance of our approach, especially for realistic scenarios that perform multiple alignments per pair of input reads. While the HPC systems offer superior performance to the cloud, the performance across 4 platforms

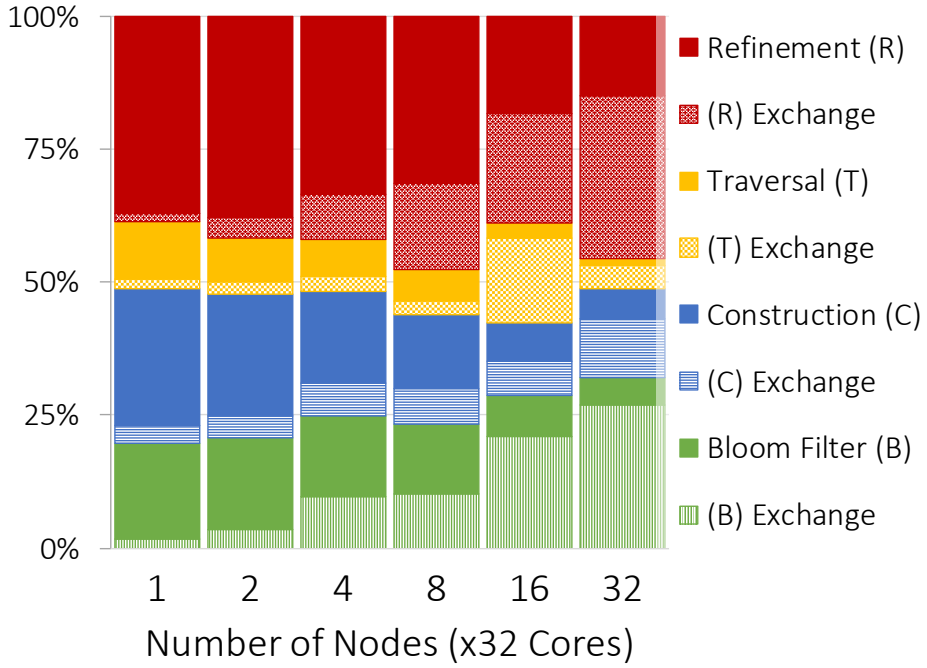


Figure 4.10: Strong scaling runtime breakdown on Cori with minimum computational-intensity workload (*E. coli* 30×, one-seed). The **y-axis** is percent of total runtime.

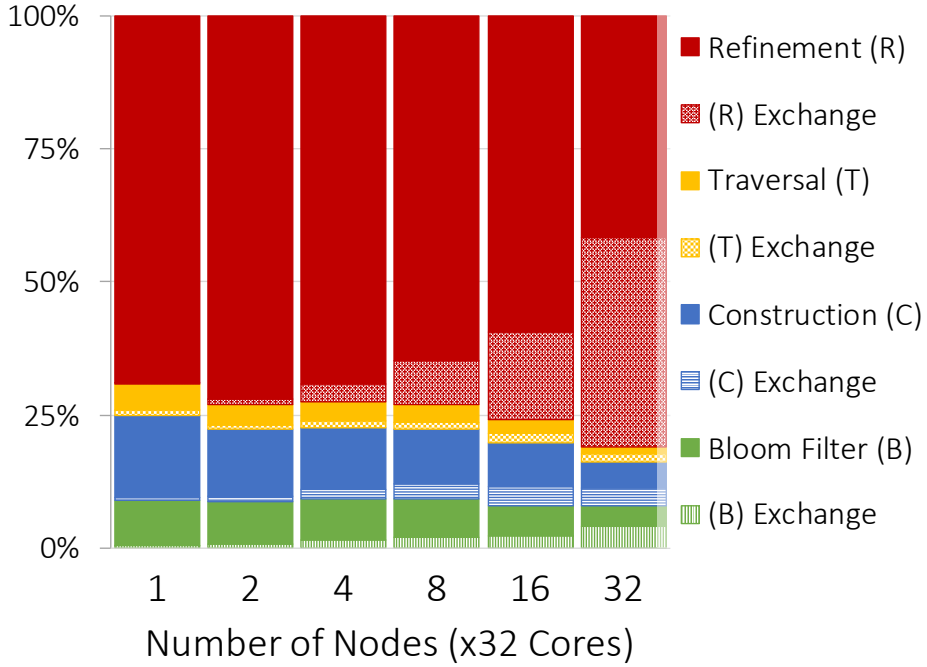


Figure 4.11: Strong scaling runtime breakdown on Cori with the *E. coli* 100× all-seeds $q = 1$Kbps workload, the input size of which is 3.5× larger and computes roughly 20× more alignments than the workload of Figure 4.10. The **y-axis** is percent of total runtime.

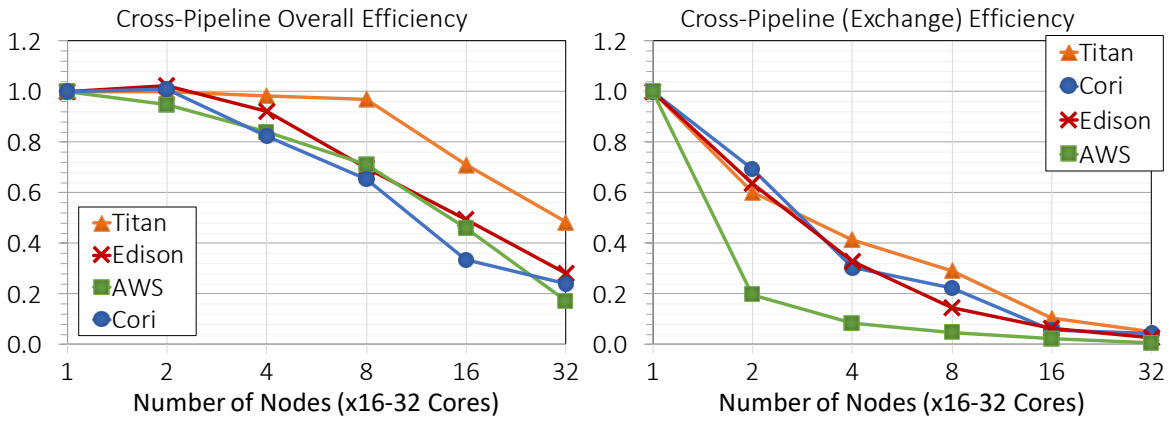


Figure 4.12: Efficiency of the whole pipeline for all architectures, strong scaling with minimally compute-intensive workload (*E. coli* 30× one-seed). Overall efficiency is shown on the left and efficiency of the exchanges is shown on the right. Efficiency (y-axis) in both is calculated relative to single node performance.

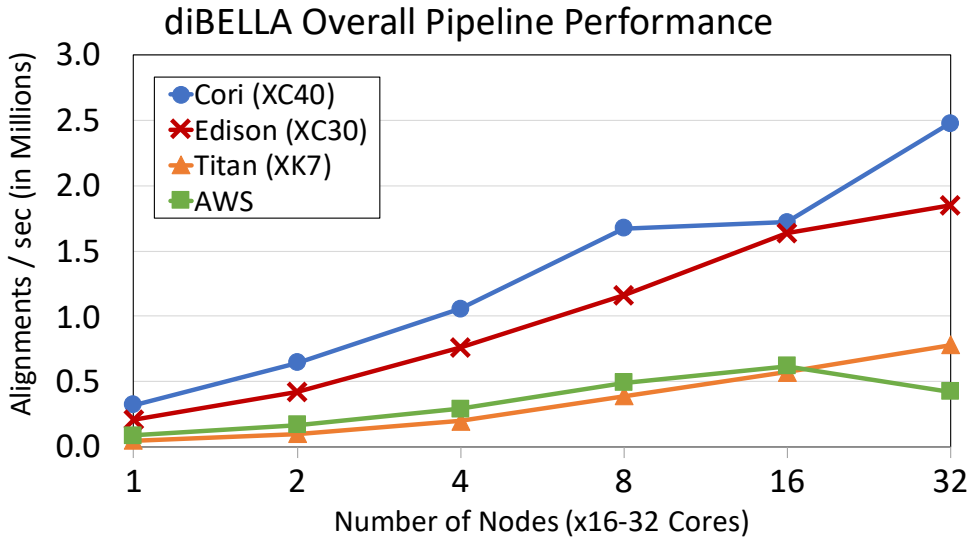


Figure 4.13: Cross-architecture strong scaling performance of the pipeline as a whole, in terms of millions of alignments per second given the *E. coli* 30× one-seed workload.

benefits from the multi-node parallelization. The application is dominated by irregular all-to-all (many-to-many) style communication and the study reveals some of the performance anomalies on particular systems, as well as general scaling issues at larger node counts. We believe that in addition to being a useful tool for bioinformatics, either standalone or as part of a larger pipeline, DiBELLA also represents an important parallel workload to understand and drive the design of future HPC system and communication libraries. It is also a foundation for future optimizations in single node pairwise alignment, that retains the efficiency of the sparse interactions.

The empirical evaluation presented simulates anticipated changes in long read overlap hypergraph analysis workloads over time. We anticipate the computational intensity, specifically of the many-to-many alignment, to be reduced by improvements in three directions. First, we expect improvements in overlap detection and seed selection (reducing the number of alignments computed). Second, we expect ongoing improvements in pairwise alignment algorithms, as an old yet vibrant area of research. Third, we expect improvements in pairwise alignment software, including hardware accelerator utilization among other optimizations. Since the original publication of these results [13], the first GPU implementation of the X-drop algorithm was published in our related work [66] (See Chapter 6). Our simulation across workloads varying computational intensity reveals that the bottleneck of the pipeline is the alignment stage, and within the alignment stage, the scalability bottleneck is the many-to-many read exchange. The analysis also reveals the opportunity to overlap communication and computation, even under computational minimal settings (see Figures 4.10-4.11).

To study computation and communication balance, the study has thus far set aside memory. The next chapter will reintroduce memory requirements into the overall study, and highlight that memory capacity is not an independent consideration for scalable design application design. Thus, the next chapter presents and evaluates an alternative asynchronous approach overlapping communication and computation while minimizing memory footprint.

## 4.7 Future Work: Overlap Hypergraph Data Analysis

The output of DiBELLA, right before or after refinement, can be used to analyze the underlying overlap hypergraph. This may be useful for future partitioning and load balancing optimizations, as well as for biological analysis. Overlap data collected with DiBELLA for the *E. coli* 30× workload is shown in Figures 4.14-4.15. The data is plotted as a read-by-read matrix; the numbers on the x-axis and y-axis correspond to reads. Non-zeros indicate an overlap between the respective read pair. Note, only the upper/lower triangle of the logically symmetric matrix is plotted; that is, for each overlap $(a, b)$ between read $a$ and read $b$, only $(a, b)$ or $(b, a)$ is plotted (not both). The total number of non-zeros over the number-of-reads-squared is 0.008. In order for the non-zeros to be visible in the plot, the dots marking non-zeros are disproportionately displayed. In Figure 4.14, reads are numbered according to an essentially random ordering; there is little to no visually identifiable structure. In Figure 4.15, reads are numbered by their relative lengths, increasing left-to-right and top-

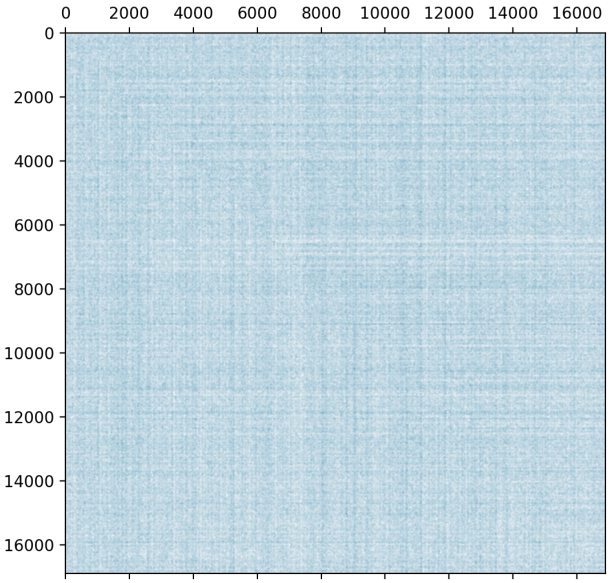


Figure 4.14: Long-read by long-read matrix dot plot. Non-zeros are overlaps discovered with DiBELLA. The number of non-zeros over the number of reads squared is 0.008. Reads are numbered according to an essentially random ordering of the reads.

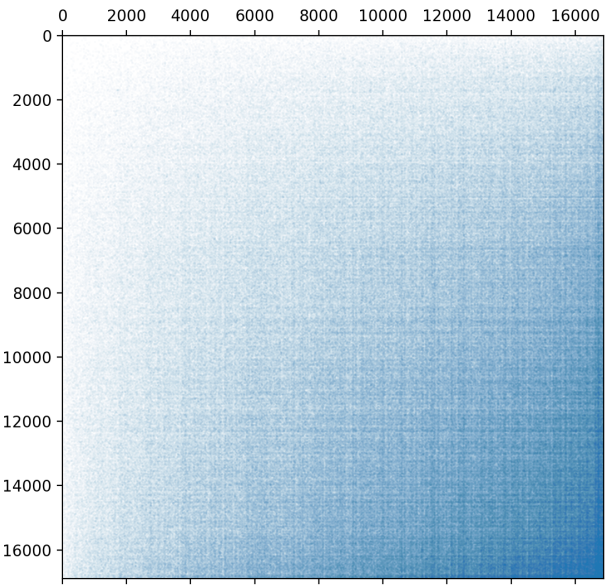


Figure 4.15: The data in Figure 4.14 re-plotted: reads are numbered by their relative lengths, increasing left-to-right and top-to-bottom. The number of overlaps per read increases with the length, but not perfectly linearly.

to-bottom; the number of overlaps per read appears to increase with the read length, as we might expect. While we have made several other observations from analyzing this and other data sets, analyzing such data is wide-open for future research. DiBELLA is a useful tool for collecting overlap hypergraph data across small and large genomes, with a flexible set of parameters for adjusting how overlaps are detected.

# Chapter 5

# Asynchronous versus Bulk Synchronous Overlap Hypergraph Refinement

In distributed computing models, memory capacity is often considered an orthogonal concern to computation and communication performance. As demonstrated for the applications in Chapter 3, however, there are trade-offs between memory and computational and communication resource utilization in practice. This chapter will consider these trade-offs for distributed long read overlap hypergraph refinement.

Long read overlap hypergraph refinement is a challenging problem for distributed memory parallelization due to its highly irregular nature, as described in Chapters 2 and 4. Chapter 4 examined the computational and communication balance for this problem; and the empirical evaluation therein isolated these components from memory consumption rates, for the purposes of that study. This chapter describes the memory requirements, and highlights the interdependence of balancing memory, computation, and communication for successful parallelization of long read overlap hypergraph refinement.

To begin the discussion, we provide upper and lower bounds on aggregate space requirements, relative to the characteristics of our hypergraph formulation from Chapter 2. In contrast to bulk-synchronous approaches, asynchronous approaches have the potential to meet the space lower bound, while hiding the communication latency of the fine-grained communication necessary for doing so. In general, either approach can work within the limits of available memory, however. Therefore, we describe and implement the necessary extensions to the bulk-synchronous approach introduced in Chapter 4. We also introduce a basic asynchronous algorithm and implementation thereof in UPC++. Our empirical evaluation uses three workload configurations to establish comparability, and to examine the performance gap between the two, under memory limitations determined by real working data set sizes.

While we demonstrate the potential of asynchronous approach in this process, we highlight the path to achieving scalability under each approach. We leave more extensive cross-

architectural analysis to future work. Ultimately, this study contributes to the broader discussion of asynchronous versus bulk-synchronous parallelism from an application-grounded standpoint.

## 5.1 Introduction: Challenges and Opportunities

Before describing challenges and opportunities in the following subsections, we briefly recapitulate the overlap hypergraph abstraction from Chapter 2, with notation that will be helpful for the following analyses. In Chapter 2, we formulated the long read many-to-many alignment problem as the complete traversal and refinement of an irregular hypergraph, $G' = \{V, H, \vec{\psi}, \vec{w}\}$. Each vertex in the hypergraph, $v_{(a,b)} \in V \mid (a,b) \in H$, must be visited (the respective pairwise alignment computed) to both refine the hypergraph (remove vertices depending on the pairwise alignment score) and to produce precise alignment information for retained vertices. In order to compute the pairwise alignment represented by vertex $v_{(a,b)}$, the information in the hyperedges $a$ and $b$ (reads) are required. The variability in the lengths or sizes in memory of the reads is represented with weight vector $\vec{w}$, such that $\forall h \in H \; \exists \; w_h \in \vec{w}$. For later convenience, we use $l_{avg}$ to denote the average read length $l_{avg} = \Sigma\vec{w}/|H|$. The variability in pairwise alignment computational costs is similarly represented by a weight vector $\vec{\psi}$ such that $\forall \; v \in V \; \exists \; \psi_v \in \vec{\psi}$. We will use this formulation to establish analytical performance bounds in terms of the characteristics of a given overlap hypergraph throughout the chapter.

### Memory Footprint Minimization

Minimizing memory footprint is a priority for supporting application runs across a broad range of distributed architectures and user constraints (limited compute resources). These include small distributed memory architectures and clusters, and architectures (large or small) in which the ratio of memory to compute resources is relatively low; it includes not only "skinny" node architectures but also "fat" node architectures in which fully utilizing the compute resources severely limits per-core memory, such as on various multi- and many-core architectures. The experimental evaluation in this chapter includes one such many-core architecture, Cori KNL at NERSC.

While the bulk-synchronous algorithm for communicating reads (cut hyperedges in the partitioned hypergraph), presented in previous chapter, maximizes bandwidth utilization, it also maximizes memory footprint. The bulk-synchronous read exchange requires memory for all replicated reads up front, at both the target and destination. Recall, the original algorithm assumes nothing about the order of reads in the input. The partitioning strategy is data-independent, in the sense that partitions are computed with respect to size of the input data in memory and no other characteristic of the data. The pairwise alignment computations (vertices in the hypergraph, $V \in G'$) are distributed with respect to the existing partitioning of the reads (hyperedges, $H \in G'$). The invariant that every vertex $v_{(a,b)} \in V$ is mapped to a

partition containing either $a \in H$ or $b \in H$ is preserved. Furthermore, vertices are distributed across the given $P$ processors such that each processor's partition contains roughly the same number of tasks, $\approx |V|/P$. In the worst-case, there is exactly one cut of a hyperedge for every vertex $v \in V$.

Equation 5.1 is an upper bound on the growth of the aggregate memory requirement relative to the number of characters in the input ($\Gamma \cdot d$) and the vertex set size, $|V|$. The constant $c_1$ denotes the memory cost of representing and alignment task in memory while $l_{avg}$ denotes the size of a read with respect to the average length of reads in the input. The $(2 \cdot l_{avg})$ term follows from the fact that we must allocate space at both the target and the destination of any replicated and communicated read. The size of the vertex set will depend on the input and on the overlap detection algorithm.

$$\Gamma \cdot d + |V|c_1 + |V| \cdot 2 \cdot l_{avg} = O(\Gamma \cdot d + |V|(l_{avg})) \tag{5.1}$$

This is a problem in practice because the aggregate minimum memory requirement (for communication alone) may be many times larger than the memory required for the input reads alone, based on the size of the (not yet refined) overlap hypergraph. A simple solution is to reduce the bulk-synchronous superstep sizes such that the memory required for incoming and outgoing message buffers is equal to that available. This strategy reduces bandwidth utilization and also incurs the overhead of multiple many-to-many exchanges.

$$\Gamma \cdot d + |V| \cdot c_1 + P \cdot l_{avg} \cdot c_2 = O(\Gamma \cdot d + |V| + P \cdot l_{avg}) \tag{5.2}$$

Alternatively, an asynchronous approach can not only hide communication latency of retrieving reads, but can also meet the aggregate space lower bound shown in Equation 5.2. The lower bound follows from the fact that each parallel processor needs to store a constant number of additional reads at any given time for all $P$ processors to compute all alignments involving remote reads. The constant $c_2$ in the lower bound is 2, one for a remote read required for local computation, and one read replicated from the local partition in response to a request. Message aggregation optimizations may be included in the model by increasing $c_2$.

In the following, we will focus on a bulk synchronous strategy that adjusts superstep sizes to available memory, and on an asynchronous strategy that simply meets the space lower bound. We will return to the topic of message aggregation and other optimizations, after introducing the basic asynchronous algorithm and implementation, which will serve as a baseline in our empirical evaluation. Note, while we focus on these two approaches, they form the basis of hybrid approaches as well.

## Communication-Computation Overlap

The bulk-synchronous algorithm for communicating reads and computing alignments (traversing the hypergraph of alignments in parallel) in Chapter 4 does nothing to hide the overhead of communicating reads (cut hyperedges). Some workloads are so dominated by the pairwise

alignment computation (e.g. by 97-99%, Figure 4.2), that even this communication overhead is a negligible component of the overall runtime. However, with improvements in the pairwise alignment computation or at higher scales, the exposed communication overhead can be significant. The results in Figure 4.11 , shows a runtime breakdown of the pipeline's strong scaling performance on a workload with limited computational intensity. At 32 nodes (1,024 cores), the communication overhead of the alignment stage is one of the two dominant bottlenecks; the computation of pairwise alignments is the other. At all scales, the runtime breakdown shows there is enough computation to completely hide the communication overhead, even with computational intensity limited to an extreme by runtime settings. In general, the opportunity for computation and communication overlap depends on the average time to compute a batch of pairwise alignments involving a given remote read $a$ relative to the communication latency for retrieving read $a$.

## 5.2 A Baseline Asynchronous Many-to-Many Alignment Algorithm

We present a basic algorithm for asynchronously communicating reads and computing alignments. It will serve as a baseline for our theoretical and empirical performance analyses. The single program multiple data (SPMD) algorithm begins with each parallel processor organizing the alignment computations (a.k.a. tasks) assigned to it. Recall, task assignment from the previous chapter preserves the invariant that $\forall v_{(a,b)} \in V$ either read $a \in H$ or read $b \in H$ is local, or both $(a, b)$ are local to the processor that will perform the task $v_{(a,b)}$. In our asynchronous algorithm, each task involving a remote read $b$ and local read $a$ is indexed under $b$; those tasks for which both reads are local may be indexed under either. Once the tasks are organized by remote read, the algorithm proceeds by issuing a one-sided request for each remote read. A callback is attached to the request; once a remote read $b$ arrives, all alignment computations involving $b$ are executed. As in the original bulk-synchronous algorithm, only those alignments which meet or exceed the user or default scoring criteria are saved for output.

The author implemented this algorithm in UPC++, a C++ library for high-performance asynchronous communication and computation. Over other asynchronous languages and libraries, UPC++ supported by GASNet-EX [5] claims lower one-sided message latencies and better programmability [2]. Given that read lengths are highly variable and that, therefore, the *number* of reads per parallel processor is non-uniform, our implementation employs UPC++ remote procedure calls (RPCs) to lookup and return reads from remote data structure partitions. This avoids additional layers of indirection or additional auxiliary data structures for lookups. Furthermore, the use of RPCs in UPC++ ensures read requests from remote processors are answered one-at-a-time, avoiding potential memory overflow from read replication for multiple simultaneous requests. We also note, it has been demonstrated [27] that RPCs can outperform remote direct memory accesses (RDMA) for large messages and

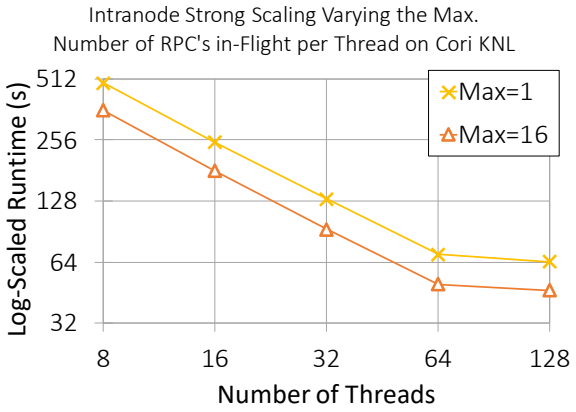


Figure 5.1: Strong scaling performance of the asynchronous algorithm implemented in UPC++, varying the maximum number of RPC's in-flight (per thread, UPC++ rank). Concurrency in terms of the number of threads increases along the x-axis. Threads are pinned to cores, except, at x=128, they are pinned to hardware hyperthreads, up to 2 per core. The overall runtime in seconds is shown on the y-axis (lower is better).

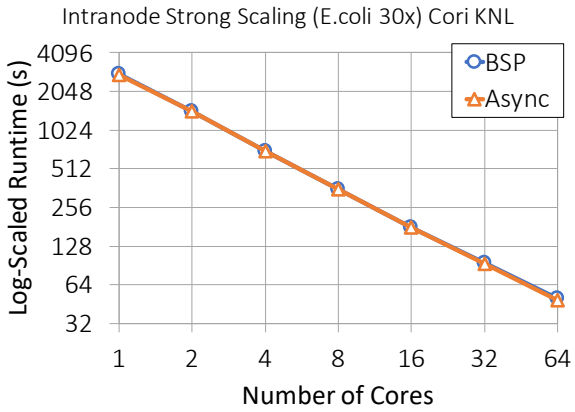


Figure 5.2: Strong scaling performance of the asynchronous algorithm implemented in UPC++ alongside the performance of the bulk synchronous algorithm implemented in MPI 2.0. Performance is shown as overall runtime in seconds (y-axis). Concurrency in terms of the number of threads increases along the x-axis. Threads, UPC++/MPI ranks, are pinned to cores, up to 1 rank per L1 cache.

for data structure lookups that involve both an index lookup and retrieval of the data itself. This most closely matches our use-case. We leave a thorough investigation of RDMA versus RPC performance for our application to future work however. An additional consideration for implementations in general are the progress guarantees of the underlying language and runtime regarding RPCs and callbacks. In our UPC++ implementation, regular application-level polling is required to ensure read requests are answered and callbacks are processed. Beyond this, GASNet-EX ensures read requests and callbacks are delivered, under the usual assumptions about the network. The next sections examine the performance of this baseline implementation against the bulk-synchronous algorithm's implementation.

## 5.3 Empirical Results

### Intranode Performance

We establish the comparability of the two codes by examining relative performance on a single node. We collected strong scaling experimental results using a real data set small enough to process within the memory of a single node of Cori Haswell and Cori KNL. Since the scalability trends are roughly the same between the two architectures, Figures 5.1-5.2 show

the results on the architecture with higher core and hardware threads counts, Cori KNL. We explore the impact of overlapping communication and computation on node by imposing a limit on the maximum number of RPC's in-flight (per UPC++ rank) and varying the limit. The "Max=1" trendline in Figure 5.1 shows the case where communication-computation overlap is most limited. Each request for a remote read waits for a response, computes the respective alignments with the received read, and then issues the request for the next remote read. While waiting, however, it may respond to requests from other processors for reads – this is managed by the UPC++ runtime. We varied this limit until a performance plateue was reached. For the Cori KNL nodes processing this workload, the plateue was reached with a limit of 16. Figure 5.1 shows the results for 8 to 128 threads with a limit of 1 versus a limit of 16. With a limit of 16, the speedup over the performance with limit 1 is 1.4 to 1.5×. For any setting, there was only marginal benefit from employing hyperthreads (see $x = 128$ in Figure 5.1). Hereforward, we focus on the results from pinning ranks to full cores (exclusive L1 caches). An intranode strong scaling performance comparison of our asynchronous algorithm ("Async") to our bulk synchronous algorithm ("BSP") is shown in Figure 5.2. The single core runtimes are approximately the same between the two versions. Scaling within the node, the runtime for processing this workload with either version was reduced from just under 1 hour on 1 core to just under 1 minute on 64 cores. Both the bulk synchronous and the asynchronous version speed-up perfectly from 1-64 cores, with negligible communication and synchronization overhead within the node. Now that we have established the comparability of the implementations on a single node, let us turn to the multinode performance.

## Multinode Performance

We compare the asynchronous and bulk synchronous multinode performance with two real data sets. The first is the *E.coli* genome sequenced with PacBio to 100× coverage. It is small enough to process on a single node, but over 3× larger than the data set used for the intranode scalability results in section 5.3. We employ it for a strong scaling comparison in which there is sufficient per processor memory for the bulk synchronous version to exchange all reads at once, achieving its lowest communication overhead. The second data set is a human genome sequenced with PacBio Circular Consensus (CCS) technology. It is roughly 28× larger than the *E.coli* 100× data set, and the initial stages of the DiBELLA pipeline, including building the hypergraph, cannot complete their processing with fewer than 8 Cori KNL nodes. DiBELLA parameters for this analysis across workloads included setting $k = 17$ and the upper limit on *k-mer* frequencies (a.k.a. "noise threshold") according to the BELLA model [23], and limiting seed-and-extend alignment to one seed per candidate-overlap (see discussion in Section 4.4 and throughout Chapter 4). For all experiments, processors are pinned to each full core (L1 cache) on a node – there are 64 per node. Furthermore, time spent in I/O is left-out; satisfactorily scalable parallel file I/O is employed in each version but the implementations are different and file I/O is not the focus of this work.

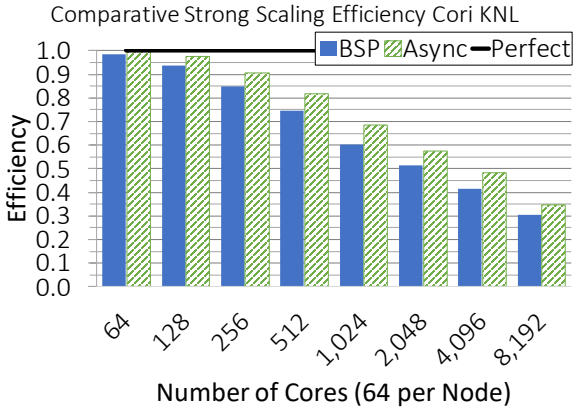


Figure 5.3: Comparative strong scaling (overall) efficiency with the *E.coli 100x* data set on Cori KNL. Efficiency (y-axis) is computed relative to single node (64 core) overall performance. The asynchronous (Async) version achieves up to 9% higher efficiency than the bulk synchronous (BSP) version.

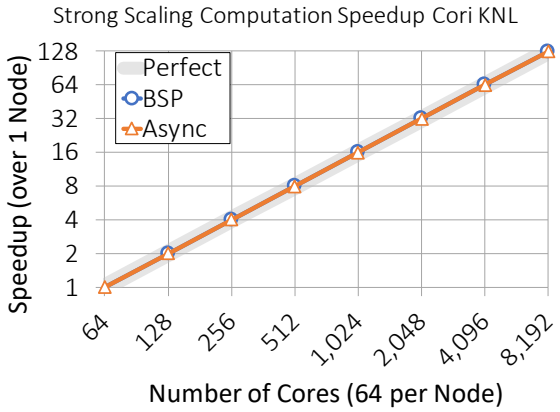


Figure 5.4: Comparative strong scaling speedup of each version's computation with the *E.coli 100x* data set on Cori KNL. Speedup is computed relative to single node (64 core) performance. The computational speedup of each version is perfect.

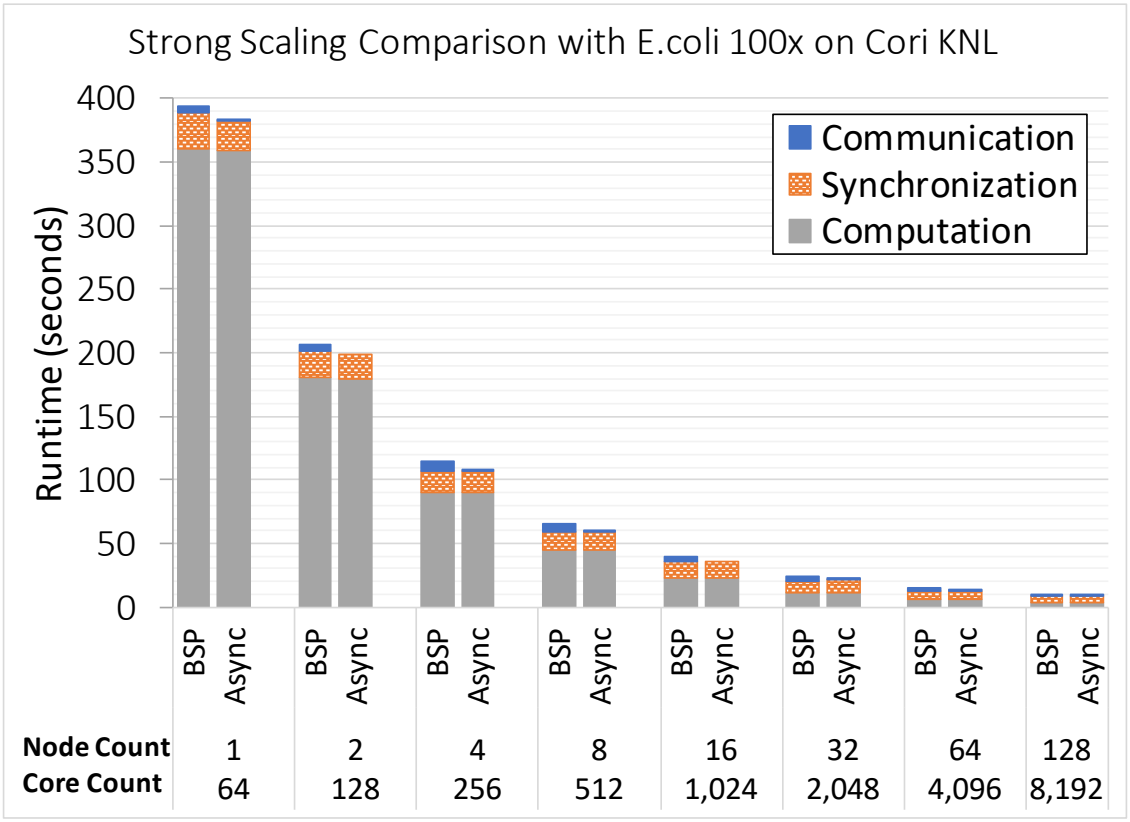


Figure 5.5: Strong scaling runtime breakdown with the *E.coli 100x* data set on Cori KNL. The bulk synchronous (BSP) version completes all read exchanges in a single round. The asynchronous version (Async) hides nearly all communication overhead.

Figure 5.3 shows the efficiency of the two implementations processing the *E.coli 100x* workload, scaling out from 1 to 128 Cori KNL nodes (64 to 8K cores). Efficiency is computed relative to the average runtime on a single node, 64 cores; we estimate it would take approximately 7 hours to process the same workload using 1 Cori KNL core. With 64 cores on 1 node, it takes roughly 6.5 minutes. Because the asynchronous version (labeled "Async" in Figure 5.3) slightly outperforms the bulk synchronous version ("BSP") on 1 node, and efficiency is compute relative to the average of the two versions on one node, the BSP efficiency is slightly lower than 1.0 at 64 cores, and the Async efficiency is slightly higher. From $1-128$ nodes, the asynchronous version achieves higher efficiency than the bulk synchronous version, up to 9%. Figure 5.4 shows the (computational) speedup of the two version; it is the same, as expected, and perfect.

We turn to the comparative runtime breakdown shown in Figure 5.5 to explain the efficiency disparity between the two versions. In both versions, the synchronization time includes the gap in time between the shortest and the longest running process, respectively, to finish all work. For the bulk synchronous version only, this includes time spent synchronizing communication rounds. The read exchange for this workload can be performed in a single round, and the additional synchronization time is negligible across scales. Between the two versions, the time spent in synchronization roughly matches; the load imbalance is the same. The communication overhead of the bulk synchronous version increases monotonically from 1% of the overall runtime on one node to 19% on 128 nodes. By contrast, the asynchronous communication overhead is not always visible and at most 0.14% between 1 and 64 nodes. At 128 nodes, the visible overhead is just under 5%. The absolute runtime of both versions on 128 nodes is just under 10 seconds; a roughly $40\times$ speedup over the single node time, and an estimated $2.5 \times 10^3$ speedup over a single core.

Figures 5.6- 5.7 show comparative strong scaling results on a larger workload, a PacBio CCS human data set. Note, the minimum number of nodes required to process this workload with our setup is 8, as determined by the memory requirement of the previous pipeline stage. We scale the workload from 8 nodes to 512 nodes (512 to 32K cores respectively). From $8-32$ nodes, there is insufficient memory for the bulk synchronous version ("BSP") to complete its read exchanges in a single round, it requires multiple exchange-compute rounds. Figure 5.6 (left), the asynchronous version is ("Async") is up to 16% more efficient under these conditions. From $64-512$ nodes, however, there is sufficient per processor memory to complete the exchange all at once. As shown in Figure 5.6 (right), the efficiency of the asynchronous version is still higher but only up to 3% higher. Figure 5.7 shows a runtime breakdown of the two versions in terms of communication, synchronization, and computation time. Similarly to the results in Figure 5.5, the pairwise alignment computation speeds-up perfectly. The synchronization time between the two versions is roughly the same across scales, as it is dominated by the load imbalance. The communication overhead of the bulk synchronous version is the source of the efficiency disparity (shown in Figure 5.6) between the two versions. The communication overhead is 17% to 34% of the bulk synchronous version's runtime across scales, while the asynchronous version completely hides its communication latency.

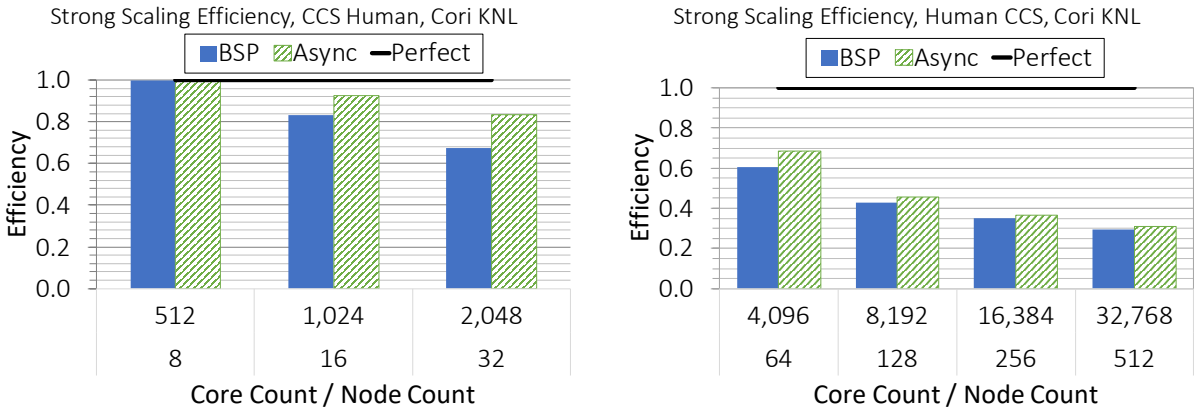


Figure 5.6: Overall strong scaling efficiency of the asynchronous ("Async") and bulk synchronous ("BSP") codes, processing the same Pacbio CCS human workload on Cori KNL. Between 8 and 32 nodes (**left**), BSP performs the many-to-many read exchanges in multiple rounds in order to stay within processor memory limits, and the Async is up to 16% more efficient. From 64-512 nodes (**right**), there is sufficient per processor memory for the bulk synchronous version to complete the exchange in a single round.

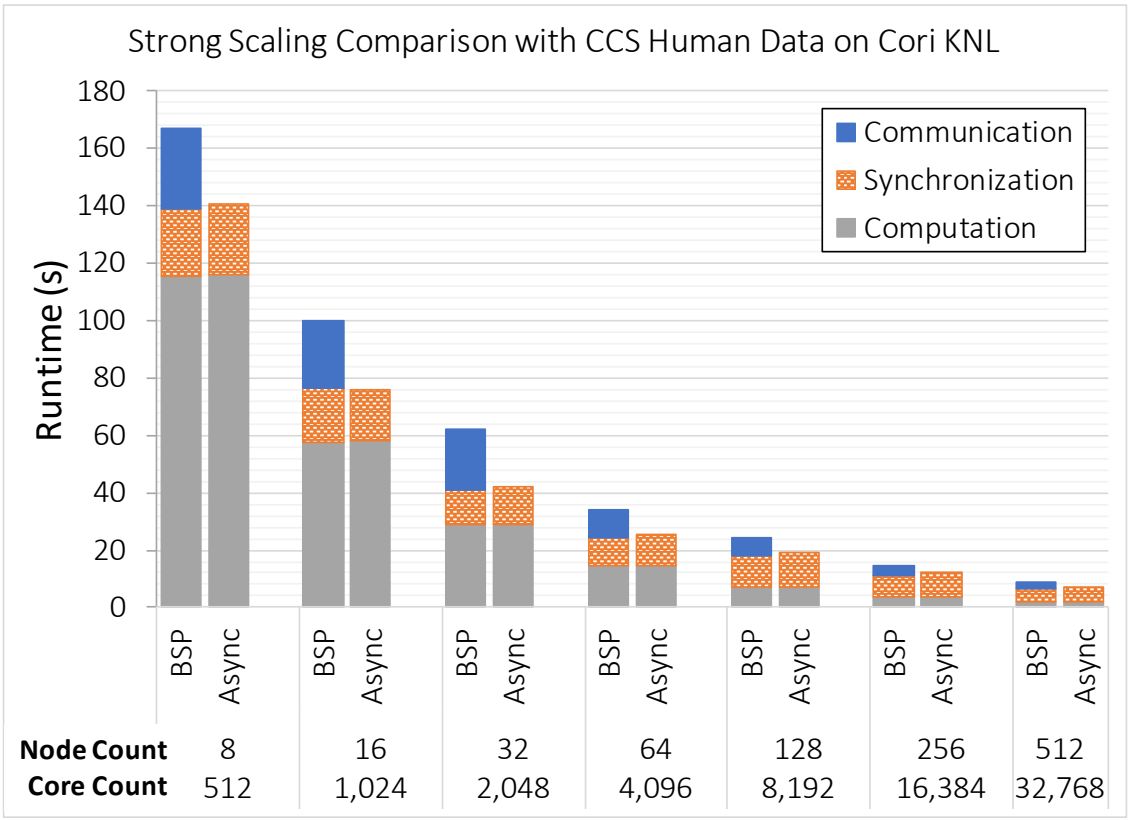


Figure 5.7: Strong scaling runtime comparison of the bulk synchronous (BSP) and asynchronous (Async) codes, processing the same Pacbio CCS human workload on Cori KNL. Async successfully hides the communication overhead.

## 5.4 Summary

This chapter examined the memory, computation, and communication trade-offs of asynchronous versus bulk-synchronous approaches to long read overlap hypergraph refinement. We described in Section 5.1 how the choice of communication algorithm for this problem is not independent from the memory footprint, and we established upper and lower bounds for memory footprint based on the size of a given overlap hypergraph. We focused on communication approaches for which the ratio of data communicated to data required by the receiving processor is exactly 1, and narrowed our focus to two main approaches. The first is an asynchronous approach that hides communication by overlapping pairwise alignment computations, and meets the space lower bound by never requiring more than one remote read at a time to make progress - though optimizations that take advantage of remaining memory may, of course, be employed in practice. The second is a simple extension of the bulk synchronous approach introduced in a previous chapter. Rather than communicating all reads at once and then computing all alignments, the extension reduces the bulk synchronous superstep size such that the communication of reads consumes no more memory than that available; a workload is processed in potentially multiple rounds of read-exchanges and pairwise alignment computations.

For comparative empirical analysis, we extended the bulk-synchronous code, presented in the previous chapter, to process workloads in multiple exchange-compute rounds depending on available memory. We also presented a simple asynchronous algorithm and its implementation in UPC++ [2]. Empirical strong scaling results were collected on Cori KNL at NERSC across three real workloads. Both codes were effective for scaling problems from a single core to all cores within a node, and from a single node to hundreds of nodes and thousands of cores. The intranode performance and scalability analysis established performance comparability of the two implementations; the performance difference was less than 3% of the overall runtime across scales. For the respective workload, our codes reduced processing time from just under one hour on 1 core, to just under one minute on 64 cores. To analyze performance differences under the best-case scenario for the bulk synchronous approach, we used a workload for which the many-to-many exchange could be performed in a single iteration, within the available memory of 1 to 128 nodes, strong scaling. The results demonstrated that even in this case, the asynchronous version was able to obtain up to 9% higher strong scaling efficiencies by hiding communication latency. Overall, our codes reduced the processing time for this workload from an estimated 7 hours on 1 core to around 10 seconds on 4K-8K cores. Finally, we analyzed the relative strong scaling performance of the bulk synchronous and asynchronous codes, processing a workload for which the many-to-many exchange could not be performed in a single exchange at small scale. The comparison highlighted the bulk synchronous approach's significant communication overhead (17 to 34%) when forced to perform multiple irregular exchanges, in order to process a workload within available memory. By contrast, the asynchronous version effectively hid all communication overhead for the same workload across all scales (8-512 nodes, 512-32K cores).

## 5.5 Conclusions & Future Work

The empirical results presented in this chapter demonstrate the great potential of asynchronous approaches for this application. However, we emphasize that the relative performance of bulk-synchronous versus asynchronous approaches is determined by the performance of communication primitives for many-to-many exchanges and by the performance of the pairwise alignment computation. Bandwidth for many-to-many exchanges supporting bulk-synchronous approaches, versus the (balance of) alignment computation to one-sided or point-to-point message latency supporting asynchronous approaches, determine the relative performance of each approach across workloads, implementations, and architectures. We further emphasize that optimizing communication and computation performance is not independent from memory capacity. Bulk-synchronous approaches trade memory for better bandwidth utilization. Asynchronous approaches can take advantage of additional memory to optimize communication and computation overlap.

On any architecture with sufficient many-to-many communication memory and bandwidth, and with single message latencies that exceed pairwise alignment computation rates, we expect the performance gap between the bulk-synchronous and asynchronous implementation to close or even switch directions. The optimizations necessary in such a setting to shift the balance toward asynchrony involve reducing the number of and increasing the efficiency of single asynchronous messages. These include data compression and batching among other possible optimizations. However, on sufficiently high-latency networks, such optimizations may still not pay-off, as in our cross-architecture cross-paradigm evaluation of HipMer [14]. In that study, the Ethernet cluster's high latency for the application's many-to-many exchange via one-sided messages overwhelmed the communication-computation overlap, even with batching and caching optimizations, causing egregious slowdowns. On the other hand, in the same settings, the bulk-synchronous code, with synchronous many-to-many exchanges, scaled linearly, remaining compute-bound across the available scales.

Furthermore, optimizations targeting just the computation will affect the overall performance of each approach differently. For bulk-synchronous approaches, we expect improvements to the computation to decrease overall runtime, and to lower the number of parallel processors at which strong scaling overall application performance becomes communication-bound rather than computation-bound. For the asynchronous approach, we expect overall runtime to improve with alignment computation optimizations until average asynchronous message latency exceeds the average pairwise alignment computation rate, at which point, communication optimizations such as those mentioned, will be necessary for any further computational optimizations to be effective.

The results from both implementations also show there is a clear opportunity to optimize load imbalance. The time to synchronize the shortest-running and the longest-running processes was visible and comparable between the codes across scales. This highlights the opportunity to optimize both the read and alignment task distributions. For all results in this chapter, we used a direct "blind" hypergraph partitioning, that avoids re-partitioning input reads and balances the (number) alignment tasks per processor with a simple heuristic,

described more fully in the previous chapter. The challenges of load balancing alignment tasks, given that the associated costs vary dynamically, were also discussed in Chapter 4. In this chapter, we focused on many-to-many communication approaches that are independent of the underlying hypergraph partitioning approach. That is, for improvements in the underlying partitioning approach, we expect the communication performance of these approaches to only improve respectively. Alternative approaches for distributing reads and alignment tasks is an open area for future work.

Finally, the codes in our empirical evaluation both follow a flat rather than a hierarchical or hybrid model. Neither code optimizes communication between processors within the same shared memory node more than the respective underlying runtime systems do by default. However, both the empirical and analytical results presented are useful for understanding performance of hierarchical or hybrid approaches as well, with simple extensions.

# Chapter 6

# Related & Ongoing Work

This chapter reviews related work, primarily with respect to distributed *k-mer* based hypergraph construction and distributed many-to-many long read alignment and overlap. Some, though not all, of the description and results included here were originally published in "DiBELLA: Distributed Long Read to Long Read Alignment" (ICPP'19) [13].

## 6.1 Distributed *k-mer* Based Hypergraph Construction

In Chapter 3, we discussed two *k-mer* based hypergraph construction applications, representing two widely accepted approaches to genomic analysis. These were de Bruijn subgraph construction for short reads, and overlap hypergraph construction for long reads. In both cases, our *k-mer* analysis includes counting the abundance of each k-mer, but also involves other metadata to store with each k-mer; implementations that purely count *k-mers* may not be comparable. There is much existing work on de Bruijn subgraph construction for genome assembly [58][40][17], and on simply *k-mer* counting and indexing [44][45][55][56][67][31][15][52][51]. However, only two primary works stand-out for offering distributed memory parallelism and scalability. The first is E. Georganas et al.'s algorithms for and implementation of de Bruijn subgraph construction in HipMer[1] for scalable genome assembly [17][18]. The second is T. Pan et al.'s general distributed memory framework and optimizations for *k-mer* counting and indexing [52][51]. The de Bruijn subgraph construction module, also referred to as the *k-mer* analysis module, in HipMer can be run standalone, and provides *k-mer* indices, *k-mer* histograms, as well as de Bruijn subgraphs. The standalone framework for *k-mer* counting and indexing by T. Pan et al., on the other hand, is designed only for *k-mer* counting and indexing. To our application case studies from Chapter 3, de Bruijn subgraph construction and overlap hypergraph construction, the additional information connected with *k-mers* is critical in the respective graphs. For de Bruijn

[1]It is noteworthy that HipMer forms the basis of the distributed memory scalable metagenome assembler, MetaHipMer [19], as well; we primarily reference the single genome assembler HipMer for simplicity.

subgraph construction, the single characters preceding and succeeding the *k-mer*, and their associated quality scores, together with the respective *k-mer* form the edges in the subgraph. For overlap hypergraph construction, as in DiBELLA, the reads associated with the *k-mers* form the hyperedges; intersecting hyperedge sets reveal potential overlap between reads. The *k-mer* histograms are useful for initial *k-mer* filtering and a valuable byproduct for analysis, but they are not a key component post-filtering. Extending and adapting optimizations proposed by T. Pan et al. for counting and indexing may be worthwhile future work. However, given that HipMer's implementation was designed with *k-mer metadata* in mind, we ultimately chose it for analysis and for extension to distributed memory overlap hypergraph construction in DiBELLA.

Comparing the hypergraph construction in HipMer and that in DiBELLA, the Bloom filter stage is identical while the hash table implementation is different. As mentioned, for each *k-mer*, HipMer need only store the two neighboring bases with their respective quality scores. DiBELLA instead needs to communicate and store information about the read where the *k-mer* originated and the location in which each *k-mer instance* appeared. Both HipMer and DiBELLA remove singleton *k-mers*, but DiBELLA also removes those *k-mers* whose occurrence exceeds the high occurrence threshold, $m$. The hash tables also represent different objects. The HipMer hash table represents a de Bruijn subgraph with *k-mer* vertices, and their connections are computed by adding the *k-mer* extensions and shifting. The subgraph is broken at points where there is no confidence in the most likely extension. The high error rates in long reads would make such a graph very fragmented. DiBELLA's hash table represents a read hypergraph with read hyperedges sharing intersecting *k-mer* sets. Pairwise seed-and-extend alignment, with *k-mers* as seeds, is used to extend and verify the region of overlap between reads. This hypergraph representation, more often referred to as the *overlap graph*[2] in the literature, is more robust to sequencing errors and thus more suitable for long-read data.

## 6.2 Many-to-Many Long-Read Overlap & Alignment

One of the biggest challenges for the analysis of sequencing data is *de novo* assembly [68], which is the process of eliminating errors and assembling a more complete version of the genome. This is especially important for plants, animals, and microbial species in which no previously assembled high quality reference genome exists. The different error rates between short and long reads lead to different approaches to assembly. For long reads, the first step is typically to find pairs of reads that overlap and resolve their differences (due to errors) by computing the *alignments*, i.e., the edits required to make the overlapping regions identical [26, 8, 9, 42, 33]. The read-to-read alignment computation is not limited to genome assembly, and is widely used in various comparisons across or within genomic data

[2]The alternative *hypergraph* formulation is introduced in this work primarily for discussing distributed memory parallelization challenges. The *overlap graph* term comes from the bioinformatics and computational biology literature.

sets to identify regions of similarity caused by structural, functional or evolutionary relationships [47]. Consequently, highly parallel long-read to long-read alignment can significantly improve the efficiency of these techniques, and enable analysis at unprecedented scales.

## Distributed Memory Parallel Approaches

Until DiBELLA, existing distributed memory alignment codes primarily targeted the alignment of a read set against a fixed, modest-sized reference sequence such as the human genome [24], where the reference can be replicated across nodes in advance. Conversely, DiBELLA computes read-to-read pairwise alignment rather than read-to-reference alignment, and distributes data across nodes for each step of the pipeline. The most similar implementation to ours, which is still quite disparate, is the end-to-end parallelization in MerAligner [21]. DiBELLA addresses long read data characteristics, and accordingly, uses a different parallelization and data layout approach. MerAligner aligns short reads to *contigs*, sequences composed of error-free *k-mers*, in order to find connections between contigs. Long reads are not only 2-3 orders of magnitude longer than short reads, but also contain errors up to 35% (versus $< 1\%$ for short reads). Hence, appropriate *k-mer* lengths for long read overlap and alignment are an order of magnitude shorter. These features combined significantly increase the size of the *k-mer* data set (see Chapter 3). Further, in MerAligner, the cardinality of the contig set is reduced significantly over the size of the input (see [18] and [14]); whereas in long-read-to-long-read alignment, a potentially all-to-all comparison of input reads may be performed. In summary, these differences result, for the long read case, in significantly higher communication, computation, and memory consumption rates, and a fundamental difference in pairwise alignment kernel and parameter choices (relating both to quality and computational cost). While exploring a PGAS design for this application similar to MerAligner's is part of ongoing work, we do not provide a direct comparison since they target significantly different problems.

## Serial and Shared Memory Parallel Algorithms & Software

In addition to BELLA, which is the method that our distributed algorithm is based on, other state-of-the-art (shared memory parallel) long-read to long-read aligners include BLASR [7], MHAP [3], Minimap2 [36], MECAT [64], and DALIGNER [49]. These codes differ in how they select pairs of reads for alignment and in the alignment algorithm. BLASR uses BWT and an FM index to identify short *k-mer* matches and then groups *k-mer* matches within a given inter-*k-mer* distance. Grouped *k-mers* are ranked based on a function of the *k-mer* frequency and highly scored groups are kept for downstream analysis. MECAT and DALIGNER use *k-mer* matches for identifying candidate overlapping pairs, similarly to BELLA and DiBELLA. MHAP computes an approximate overlap detection performing sketching on the *k-mer* set using minHash. Compact sketches are used to estimate the similarity between sequences. Minimap2 uses minimizers rather than all possible *k-mers* to obtain a compact

Table 6.1: Single node, 64 thread runtime (seconds) comparison (excluding I/O) across 3 data sets on Cori Haswell w/ 128 GB RAM. Reported DALIGNER times also exclude all pre- and post- processing.

| | *E. coli* 30x (sample) | *E. coli* 30x | *E. coli* 100x |
|---|---|---|---|
| DiBELLA | 12.74 | 65.72 | 79.45 |
| DALIGNER | 7.31 | 52.04 | 63.70 |

representation of the original read. Colinear *k-mers* on a read are chained together and used for finding possible matches with other sequences.

Like BELLA and DiBELLA, MECAT and DALIGNER use exact individual matches on longer *k-mers* to identify reads that may align well. MECAT divides reads into blocks and scans for identical *k-mers* which are used to calculate a distance difference factor first between neighboring *k-mers* hits, and then between neighboring blocks. DALIGNER computes a *k-mer* sorting based on the position within a sequence and then uses a merge-sort to detect common *k-mers* between sequences. DALIGNER supports problem sizes exceeding single node resources through a scripting frontend, that divides work into a series of independent execution steps. This script-generated-script can be executed directly (serially) on a single node, or the user can modify it to run independent batch jobs, as individual node resources become available in a distributed setting. For example, if the data is divided into blocks, $B_1, ...B_n$ then DALIGNER can be executed independently by aligning $B_1$ to itself as one job, $B_2$ to itself and $B_1$ as another, and so on. This approach addresses the memory limitations, but it is not scalable. DALIGNER's distributed memory approach reads $B_1$ from disk $n$ times and the amount of work varies significantly across nodes. Given these differences, we do not provide a direct multi-node comparison, however for completeness, we provide a single node runtime comparison of DiBELLA and DALIGNER in Table 6.1. We exclude I/O time from each measurement, since each tool handles both input and output in significantly different ways. For DALIGNER, we additionally exclude all pre-processing (initializing the database and splitting it into blocks) and post processing (all commands for verifying and merging results) time, and implicitly, the time required of DALIGNER users to extract human-readable results of interest from the database. Table 6.1 shows that DiBELLA's single node runtime is competitive with DALIGNER's across these data sets, even excluding DALIGNER's I/O, preprocessing, and postprocessing.

Of the alternatives, BELLA is the latest (and ongoing) work, with a comprehensive quality comparison to all the above [23]. BELLA's quality is competitive, especially excelling in comparisons where the "ground truth" is known. Further, BELLA offers a computationally efficient approach, yielding *consistently* high accuracy across data sets, and a well-explained and supported methodology. The quality produced by DiBELLA is at least that of BELLA (see [23] for quality comparisons over data sets also used in this study), and higher when using less restricted sets of seeds for the same workloads [23].

*De novo* genome assembly depending on long-read alignment is becoming a crucial step

in bioinformatics. Current available long-read *de novo* assemblers are Canu [33], which uses MHAP as long-read aligner, Miniasm [35] which uses Minimap2, and HINGE [28] and FALCON [9], which use DALIGNER. Flye [32] uses the *longest jump-subpath approach* [38] to compute alignments. From a hardware acceleration standpoint, there has been increasing interest in the long read alignment problem as well, as evidenced by recent work on hardware acceleration of filtering and similarity-search [60][1]. Though we do not explore it in this work, we leave it as a promising future direction.

## 6.3 Efficient Long-Read Pairwise Alignment

In Chapter 2, we defined the pairwise alignment problem and seed-and-extend pairwise alignment. Here, we discuss the specific algorithm for seed-and-extend pairwise alignment employed by BELLA and DiBELLA, its particular advantages for *k-mer*-seeded long-read to long-read alignment, and recent related work. Distributed memory scalable many-to-many alignment is the larger focus of DiBELLA, indeed the software is designed for modular substitution of pairwise alignment kernels. However, the properties of the alignment algorithm used to align each pair of reads in the many-to-many alignment is relevant to understanding computational intensity and to future related work.

For the long-read to long-read alignment problem, a case study of this dissertation, and for long-read to reference alignment as well, seed-and-extend alignment has been shown to be particularly practical, given their lengths (roughly $10,000 - 100,000$ characters) and the quadratic cost of exact pairwise alignment with dynamic programming [59][50]. It has also been shown to be sufficiently accurate in the presence of high, long read error rates. For methodologies which carefully choose seeds for extension, it may be more accurate than alternatives, such as, for example, searching for globally optimal alignments [50]. Several notable long read aligners were developed using the seed-and-extend paradigm, including BWA-SW [37], Bowtie2 [34], and GASST [54] among others. BELLA and DiBELLA improve upon these ideas and are specifically designed for many-to-many long read alignment.

Many different forms of seeds are proposed in the literature, that vary by how many and which types of errors should be permitted in the seed, the optimal lengths of seeds relative to the given string pair, whether the length should be fixed or adaptive and so on. Some of these are *maximal exact matches* (MEMs), *maximal unique matches* (MUMs), adaptive seeds [29], $q$-grams[53], spaced or gapped seeds [43][39]. The most popular form of seeds in the literature are fixed-length exact-matching *k-mers*. This is the type of seed used in the development of BELLA[23] for long read to long read alignment, and also currently employed by DiBELLA [13].

### X-drop Seed-and-Extend Pairwise Alignment

The X-drop algorithm [69] is a seed-and-extend approach to pairwise alignment that can achieve average-case linear complexity. It does so by terminating the alignment search early

if the difference in score between the best alignment discovered and the alignment actively being explored falls below a certain threshold, $X$. Thus, it restricts the search space, focusing only on relatively high scoring alignments extended from the given seed. With an appropriate scoring matrix and value of $X$, the authors of X-drop have shown the algorithm can produce the same alignments as exact dynamic programming algorithms [69]. The BELLA [23] methodology, upon which the distributed memory parallel software DiBELLA is based, not only defines a procedure for detecting probabilistically correct *k-mers* to use as seeds, but also the procedure for dynamically choosing a final scoring threshold, such that the probability of discarding a true overlap is minimized. The original versions of BELLA and DiBELLA employ an efficient C++ CPU implementation of X-drop from the SeqAn library [10]. The next section discusses GPU accelerated alternatives.

## Parallel Pairwise Alignment

Though long read pairwise alignment is costly relative to common string operations like hashing, sorting, and even short read pairwise alignment, it does not involve sufficient work or data for shared memory parallelization, and certainly not for distributed memory parallelization. Finer grained parallelism such as SIMD or GPU parallelism is possible, though even for GPUs, multiple pairwise alignments are typically batched.

There is a vast amount of literature regarding short read pairwise alignment acceleration, but only recently has long read pairwise alignment drawn attention. Existing approaches focus on accelerating the exact dynamic programming approaches of Smith-Waterman [59] and Needleman-Wunsch [50] and slight variants thereof. Among those for which long read alignment support is possible, two primary works stand out, CUDASW++3 and MR-CUDASW++. CUDASW++3 [41] combines SIMD instructions and GPU parallelism to accelerate the Smith-Waterman algorithm. Unlike many tools designed and optimized for short reads, it can align sequences longer than the typical short read lengths of 100-250 characters. However, its performance for sequences longer than 400 characters is relatively much worse. Additionally, when utilizing only the GPU and not also SIMD instructions, their maximum attained performance is roughly 1/3 of their peak performance. Building on this work, A. Muhammadzadeh presented MR-CUDASW++ [48]. MR-CUDASW++ was inspired by CUDASW++3 but optimized for "medium length" reads. A. Muhammadzadeh compares MR-CUDASW++ to other tools, with CUDASW++3 as its closest contender, across sequences lengths of $10^3$, $10^4$, and $10^5$. MR-CUDASW++ achieved speedups of $1 - 2\times$ over CUDASW++3.

While hardware acceleration efforts can reduce the practical runtime of Needleman-Wunsch and Smith-Waterman, they do not change the algorithmic complexity from $O(n^2)$, where $n$ is the length of the longer read. Alternative approximate algorithms, such as X-drop, restrict the alignment search space in order to achieve near linear average-case complexity. However, because of the dynamic branching used to restrict the search space, they are much harder to parallelize on GPUs. The potential payoff of successfully accelerating an $O(n)$ algorithm versus an $(n^2)$ algorithm is significant for long read to long read alignment, since

$n$ for long reads can be around $10^5$ characters. Furthermore, BELLA and DiBELLA perform many such pairwise alignments with an average-case $O(n)$ rather than an $O(n^2)$ algorithm. Therefore, in work related to but not included as part of this dissertation, we presented a GPU adaptation and implementation of the X-drop algorithm named Logan in IPDPS'20 [66]. As far as we are aware, this is the first and latest attempt to accelerate the X-drop algorithm on GPUs. The most closely related and recent work is by Feng et al. [16] and accelerates the *seed-chain-extend* pairwise alignment algorithm in minimap2 [36]. The algorithm is quadratic in the length of the reads when computing traceback and linear otherwise. Both represent growing interest and progress in this area.

Part of ongoing and future work is integrating accelerated pairwise alignment into the larger distributed memory pipeline. As highlighted by this dissertation, particularly for many-to-many long read alignment in Chapters 4-5, balancing computation with communication, and communication design with memory will remain the main challenge. We expect the integration of GPU acceleration to affect not only the computation, but also memory consumption and management, particularly for read replication and movement, and the communication optimality.

## 6.4 Multiple Sequence Alignment

The overlap detection described in Chapters 2-4, finds sets of (at least) two reads that all potentially overlap. As defined in Chapter 2, optimal multiple sequence alignment minimizes/maximizes the alignment score of a set of two or more sequences, so why do we employ pairwise alignment rather than multiple sequence alignment? In the initial overlap verification, performing multiple sequence alignment on a set of reads that do not certainly overlap may produce poor alignments for subsets that truly overlap, because it would align those that truly overlap to others that may not belong in the set in the first place. Once a subset of reads has been verified to overlap with pairwise alignment, it would be reasonable to incorporate multiple sequence alignment in downstream analyses. We leave additional downstream analyses to future work however, and also note, pairwise alignment can be used to approximate multiple sequence alignments.

# Chapter 7

# Conclusions & Future Work

Several patterns emerge across our case studies and in comparisons of alternative parallelization. These include many-to-many computation and communication, domain-specific filtering, hashing, and alignment. A discussion of all of these patterns, as they appear in an even wider range of bioinformatics applications, is now available [65]. The predominant pattern, from the lens of this study, is a kind of sparse, irregular many-to-many computation and communication.

For any given workload, the exact many-to-many pattern, the data exchange volume, and the computational work are determined by characteristics of the data that are not known until runtime. To construct the distributed hash table, supporting scalable short read de Bruijn subgraph construction and long read overlap hypergraph construction in Ch. 3, load balancing *k-mers* from an unknown distribution requires a many-to-many *k-mer* exchange. Redistributing alignment tasks requires a many-to-many exchange of task - represented by pairs of read identifiers and seed locations, as described in Ch. 4. Long read overlap hypergraph refinenment, which labels vertices with exact alignment information, and also removes vertices for which alignment scores are too low, requires a many-to-many exchange of the reads is required to complete the pairwise alignment computations, as described in Ch. 4-5. The pattern of all of these exchanges and the associated computation depends on unknown structures in the data, and discovering those unknown structures in the input data, is at least part of the goal of each runtime analysis.

The optimality of any approach to scaling these many-to-many application case studies depends on the balance of computation to communication for fundamental units of data. The fundamental unit of data in distributed hash table construction in Ch. 3, is the *k-mer*. The fundamental unit of computation, is computing and hashing each *k-mer*; the complexity thereof is $O(k)$. For the overlap hypergraph traversal that identifies candidate overlaps (pairwise alignment tasks for the hypergraph refinement) in Ch. 4, the fundamental unit of data is the information associated with potential read-to-read-overlap locations. In our case, this is two integers identifying reads, and positions of the filtered *k-mer* seeds for seed-and-extend alignment. The computation thereof reads integer pairs, stored consecutively in an array, and computes a few integer and boolean logic operations – an $O(1)$ computation. In

the case of long read overlap graph refinement, the fundamental unit of data are the reads, and the computation is pairwise alignment of read pairs. As described in Ch. 2, pairwise alignment in general is $O(l^2)$ for reads of length $l$, and average-case $O(l)$ for the X-drop seed-and-extend implementation employed by our work.

Identifying these units and their sizes in practice undergirds the respective performance models and their parameters, performance models that can inform application design and optimization across bulk-synchronous and asynchronous paradigms. In Ch. 3 on short read de Bruijn and long read overlap graph construction, we highlighted that the size of $k$ varies by application. For short read *k-mer* analysis, $k$ may be in range [30, 150] - more than half of short read length. For long read *k-mer* analysis, $k$ is typically in range [14, 19]. For metagenome *k-mer* analysis, $k$ may range within [11, 21] and vary during the runtime analysis. With this knowledge of practical $k$-values, we intuit that the O($k$) computation on *k-mers* is unlikely to balance out the cost of communicating *k-mers* (individually) across nodes in the many-to-many *k-mer* exchange. Similar intuition follows for our overlap hypergraph traversal. For overlap hypergraph refinement on the other hand, we know expected long read lengths are relatively large, in the range of $[10^3, 10^5]$ characters. Our asynchronous approach to refinement in Ch. 5 successfully hid the asynchronous communication costs, because the computational cost of pairwise alignment was balanced against the cost of asynchronous read retrieval. This was supported in part by high-performance UPC++ RPCs and Cori KNL's low-latency interconnect. With changes in communication primitives, alignment kernels, interconnect performance, *et cetera*, these costs may need to be re-balanced for the success of the same approach.

While the fundamental units of data that determine the computational complexity and communication volumes of these workloads are key to application performance, so to are architecturally-determined communication and computation rates. There is a huge variety of communication primitives that can be used to implement many-to-many exchanges in practice. We have shown in other work the usefulness of one-sided put and get, with and without atomic synchronization, for implementing the many-to-many communication patterns in our applications [14][22]. In that work and in this, we also employ variable all-to-all MPI for bulk-synchronous many-to-many exchanges. Remote Procedure Calls, point-to-point message passing, and various forms of batched-asynchronous messaging, such as MPI 3.0's, are also alternatives. Measuring the cost of such communication primitives relative to computational costs on target platforms can usefully narrow the design and optimization space for scalable many-to-many applications. For the extremes of one-at-a-time asynchronous and all-at-once bulk-synchronous approaches to many-to-many communication, the key questions for future performance models are the following. For the one-at-a-time asynchronous extreme, the key questions are: what is the performance of each fundamental unit of computation and what are the latencies of respective asynchronous communication primitives, are they balanced? While they are balanced, we can expect asynchronous approaches to scale successfully. Optimizations targeting just the computation or just the communication will have limited impact if the balance is neglected, and further, ignoring memory utilization can yield designs that are unrealizable for a given data set and hardware platform. For the

bulk-synchronous extreme, the key questions are: how does bandwidth (particularly irregular all-to-all or many-to-many bandwidth) scale, does it keep up with the scalability of the target application's computation? In Ch. 4, scaling-out the bulk-synchronous implementation using MPI_Alltoallv, we saw clear computation-communication cross-overs for 3 out of the 4 platforms in the study, and computation rates balanced with the communication performance across scales on only 1 platform, under specific conditions. As pointed out there, the MPI_Alltoallv exchange scaled poorly for the highly irregular many-to-many exchanges. It highlights optimization opportunities for libraries, runtimes, and architectures.

As discussed throughout Ch. 3-5, memory capacity cannot be considered independently from computation and communication in parallelizing these data-intensive many-to-many applications. Bulk-synchronous approaches effectively trade memory for bandwidth utilization; achieving optimality under these approaches depends on sufficient available memory resources. For asynchronous approaches with computation costs less than the associated communication costs, lowering or amortizing the communication cost per computational unit to balance them, may require message batching, data compression, or other optimizations that are inherently space-oriented.

Alternative approaches to many-to-many computation and communication make similar trade-offs. For example, Generalized Communication Avoiding (CA) [30] techniques target applications with similar patterns, such as N-Body problems; albeit, these have only been analyzed for the simpler "dense" case where the all-to-all interactions involve all pairs, not a filtered subset, and the computational costs and data volumes are assumed equal across tasks and elements (bodies). They avoid communication by replicating state upfront, before the pattern of computation and communication is known. The replicated data is a superset of the data that will be needed by local processors for the computation. For arithmetically-intense applications or workloads that under-utilize memory, this is an effective strategy. Employing CA techniques at-scale with respect to the workload size, if extended to the irregular case, is a promising direction for future work.

The fundamental units of data, computation, and communication, and the trade-offs between memory, computation, and communication, appear across the case studies and architectures in this work. All of these components form the basis of a performance model for data-intensive irregular applications, characterized by the many-to-many motif. Such models are essential to software-hardware codesign, supporting the scalability of this challenging class of application. Because such important applications, such as those we have studied for genomics, fall into this class of applications, this work is broadly impactful. Our future and ongoing work is generalizing and validating analytical performance models built on this work, and also on empirical microbenchmarks, extending our previously published work [22]. Together, the analytical and empirical models can broadly support effective parallelization and scalability of these important applications, and inform the design of future hardware systems and programming systems, as well as algorithm and data structure libraries.

# Bibliography


[1] Mohammed Alser et al. "Shouji: A Fast and Efficient Pre-Alignment Filter for Sequence Alignment". In: *Bioinformatics* (2019).

[2] John Bachan et al. "UPC++: A High-Performance Communication Framework for Asynchronous Computation". In: May 2019, pp. 963–973. DOI: 10.1109/IPDPS.2019.00104.

[3] Konstantin Berlin et al. "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing". In: *Nature biotechnology* 33.6 (2015), pp. 623–630.

[4] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[5] Dan Bonachea and Paul H. Hargrove. "GASNet-EX: A High-Performance, Portable Communication Library for Exascale". In: (Oct. 2018). DOI: 10.25344/S4QP4W.

[6] Massimo Cafaro and Piergiulio Tempesta. "Finding frequent items in parallel". In: *Concurrency and Computation: Practice and Experience* 23.15 (2011), pp. 1774–1788.

[7] Mark J. Chaisson and Glenn Tesler. "Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory". In: *BMC Bioinformatics* 13.1 (2012), p. 238. ISSN: 1471-2105.

[8] Chen Shan Chin et al. "Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data". In: *PLoS Medicine* 10.6 (Apr. 2013), pp. 563–569. ISSN: 1549-1277.

[9] Chen-Shan Chin et al. "Phased diploid genome assembly with single-molecule real-time sequencing". In: *Nature methods* 13.12 (2016), p. 1050.

[10] Andreas Döring et al. "SeqAn an efficient, generic C++ library for sequence analysis". In: *BMC bioinformatics* 9.1 (2008), p. 11.

[11] Dannie Durand. *Lecture notes in Computational Genomics and Molecular Biology*. Last accessed 05/22/20. 2016. URL: http://www.cs.cmu.edu/~durand/03-711/2016/Lectures/PW_sequence_alignment_2016.pdf.

[12] John Eid et al. "Real-Time DNA Sequencing from Single Polymerase Molecules". In: *Science* 323.5910 (2009), pp. 133–138. ISSN: 0036-8075. DOI: 10.1126/science.1162986. eprint: https://science.sciencemag.org/content/323/5910/133.full.pdf. URL: https://science.sciencemag.org/content/323/5910/133.

[13] Marquita Ellis et al. "DiBELLA: Distributed Long Read to Long Read Alignment". In: *48th International Conference on Parallel Processing (ICPP 2019)*. Kyoto, Japan, Aug. 2019. ISBN: 978-1-4503-6295-5. DOI: 10.1145/3337821.3337919.

[14] Marquita Ellis et al. "Performance characterization of de novo genome assembly on leading parallel systems". In: *European Conference on Parallel Processing*. Springer. 2017, pp. 79–91.

[15] Marius Erbert, Steffen Rechner, and Matthias Müller-Hannemann. "Gerbil: a fast and memory-efficient k-mer counter with GPU-support". In: *Algorithms for Molecular Biology* 12.1 (2017), p. 9. ISSN: 1748-7188. DOI: 10.1186/s13015-017-0097-9. URL: https://doi.org/10.1186/s13015-017-0097-9.

[16] Zonghao Feng et al. "Accelerating Long Read Alignment on Three Processors". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: ACM, 2019, 71:1–71:10. ISBN: 978-1-4503-6295-5.

[17] E. Georganas et al. "Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly". In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 437–448.

[18] Evangelos Georganas. "Scalable Parallel Algorithms for Genome Analysis". PhD thesis. EECS Department, University of California, Berkeley, 2016.

[19] Evangelos Georganas et al. "Extreme Scale de Novo Metagenome Assembly". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. Dallas, Texas: IEEE Press, 2018. DOI: 10.1109/SC.2018.00013. URL: https://doi.org/10.1109/SC.2018.00013.

[20] Evangelos Georganas et al. "HipMer: an extreme-scale de novo genome assembler". In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. 2015.

[21] Evangelos Georganas et al. "MerAligner: A fully parallel sequence aligner". In: *IEEE International Parallel and Distributed Processing Symposium*. Hyderabad, India: IEEE, 2015, pp. 561–570.

[22] Evangelos Georganas et al. "MerBench: PGAS Benchmarks for High Performance Genome Assembly". In: *Proceedings of the Second Annual PGAS Applications Workshop (PAW17)* (Nov. 2017).

[23] Giulia Guidi et al. "BELLA: Berkeley efficient long-read to long-read aligner and overlapper". In: *bioRxiv* (2018), p. 464420.

[24] Runxin Guo et al. "Bioinformatics applications on apache spark". In: *GigaScience* 7.8 (2018), giy098.

[25] "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm". In: *DMTCS Proceedings*. 2008.

[26] Vasanthan Jayakumar and Yasubumi Sakakibara. "Comprehensive evaluation of non-hybrid genome assembly tools for third-generation PacBio long-read sequence data". In: *Briefings in Bioinformatics* (2017), bbx147.

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs". In: *Proc. 12th USENIX OSDI*. Savannah, GA, Nov. 2016.

[28] Govinda M Kamath et al. "HINGE: long-read assembly achieves optimal repeat resolution". In: *Genome research* 27.5 (2017), pp. 747–756.

[29] Szymon M Kiełbasa et al. "Adaptive seeds tame genomic sequence comparison". In: *Genome research* 21.3 (2011), pp. 487–493.

[30] Penporn Koanantakool. "Communication Avoidance for Algorithms with Sparse All-to-all Interactions". PhD thesis. UC Berkeley, 2017.

[31] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. "KMC 3: counting and manipulating k-mer statistics". In: *Bioinformatics* 33.17 (May 2017), pp. 2759–2761. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btx304. eprint: https://academic.oup.com/bioinformatics/article-pdf/33/17/2759/25163903/btx304.pdf. URL: https://doi.org/10.1093/bioinformatics/btx304.

[32] Mikhail Kolmogorov et al. "Assembly of long, error-prone reads using repeat graphs". In: *Nature biotechnology* (2019), p. 1.

[33] Sergey Koren et al. "Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation". In: *Genome research* 27.5 (2017), pp. 722–736.

[34] Ben Langmead and Steven L Salzberg. "Fast gapped-read alignment with Bowtie 2". In: *Nature methods* 9.4 (2012), p. 357.

[35] Heng Li. "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences". In: *Bioinformatics* 32.14 (2016), pp. 2103–2110.

[36] Heng Li. "Minimap2: pairwise alignment for nucleotide sequences". In: *Bioinformatics* 34.18 (2018), pp. 3094–3100.

[37] Heng Li and Richard Durbin. "Fast and accurate long-read alignment with Burrows–Wheeler transform". In: *Bioinformatics* 26.5 (Jan. 2010), pp. 589–595. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btp698. eprint: https://academic.oup.com/bioinformatics/article-pdf/26/5/589/16896917/btp698.pdf. URL: https://doi.org/10.1093/bioinformatics/btp698.

[38] Yu Lin et al. "Assembly of long error-prone reads using de Bruijn graphs". In: *Proceedings of the National Academy of Sciences* 113.52 (2016), E8396–E8405.

[39] Yongchao Liu and Bertil Schmidt. "Long read alignment based on maximal exact match seeds". In: *Bioinformatics* 28.18 (Sept. 2012), pp. i318–i324. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts414. eprint: https://academic.oup.com/bioinformatics/article-pdf/28/18/i318/16910025/bts414.pdf. URL: https://doi.org/10.1093/bioinformatics/bts414.

[40] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. "Parallelized short read assembly of large genomes using de Bruijn graphs". In: *BMC bioinformatics* 12.1 (2011), p. 354.

[41] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions". In: *BMC bioinformatics* 14.1 (2013), p. 117.

[42] Nicholas James Loman, Joshua Quick, and Jared T Simpson. "A complete bacterial genome assembled de novo using only nanopore sequencing data". In: *bioRxiv* (2015).

[43] Bin Ma, John Tromp, and Ming Li. "PatternHunter: faster and more sensitive homology search". In: *Bioinformatics* 18.3 (2002), pp. 440–445.

[44] Guillaume Marcais and Carl Kingsford. "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers". In: *Bioinformatics* 27.6 (Jan. 2011), pp. 764–770. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btr011. eprint: https://academic.oup.com/bioinformatics/article-pdf/27/6/764/16902460/btr011.pdf. URL: https://doi.org/10.1093/bioinformatics/btr011.

[45] Páll Melsted and Jonathan K Pritchard. "Efficient counting of k-mers in DNA sequences using a bloom filter". In: *BMC Bioinformatics* 12.1 (2011), p. 333. ISSN: 1471-2105. DOI: 10.1186/1471-2105-12-333. URL: https://doi.org/10.1186/1471-2105-12-333.

[46] Jayadev Misra and David Gries. "Finding repeated elements". In: *Science of computer programming* 2.2 (1982), pp. 143–152.

[47] David W Mount. "Sequence and genome analysis". In: *Bioinformatics: Cold Spring Harbour Laboratory Press: Cold Spring Harbour* 2 (2004).

[48] Amir Muhammadzadeh. "MR-CUDASW – GPU accelerated Smith-Waterman algorithm for medium-length (meta)genomic data". MA thesis. University of Saskatchewan, Saskatchewan, July 2014.

[49] Gene Myers. "Efficient Local Alignment Discovery amongst Noisy Long Reads". In: *Algorithms in Bioinformatics*. Ed. by Dan Brown and Burkhard Morgenstern. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 52–67. ISBN: 978-3-662-44753-6.

[50] Saul B Needleman and Christian D Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of molecular biology* 48.3 (1970), pp. 443–453.

[51] Tony C Pan, Sanchit Misra, and Srinivas Aluru. "Optimizing high performance distributed memory parallel hash tables for DNA k-mer counting". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 135–147.

[52] Tony Pan et al. "Kmerind: A Flexible Parallel Library for K-Mer Indexing of Biological Sequences on Distributed Memory Systems". In: *Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*. BCB '16. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 422–433. ISBN: 9781450342254. DOI: 10.1145/2975167.2975211. URL: https://doi.org/10.1145/2975167.2975211.

[53] Kim R Rasmussen, Jens Stoye, and Eugene W Myers. "Efficient q-gram filters for finding all $\varepsilon$-matches over a given length". In: *Journal of Computational Biology* 13.2 (2006), pp. 296–308.

[54] Guillaume Rizk and Dominique Lavenier. "GASSST: global alignment short sequence search tool". In: *Bioinformatics* 26.20 (2010), pp. 2534–2540.

[55] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. "DSK: k-mer counting with very low memory usage". In: *Bioinformatics* 29.5 (Jan. 2013), pp. 652–653. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btt020. eprint: https://academic.oup.com/bioinformatics/article-pdf/29/5/652/702231/btt020.pdf. URL: https://doi.org/10.1093/bioinformatics/btt020.

[56] Rajat Shuvro Roy, Debashish Bhattacharya, and Alexander Schliep. " Turtle: Identifying frequent k -mers with cache-efficient algorithms ". In: *Bioinformatics* 30.14 (Mar. 2014), pp. 1950–1957. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btu132. eprint: https://academic.oup.com/bioinformatics/article-pdf/30/14/1950/17140212/btu132.pdf. URL: https://doi.org/10.1093/bioinformatics/btu132.

[57] Jay Shendure and Hanlee Ji. "Next-generation DNA sequencing". In: *Nature biotechnology* 26.10 (2008), p. 1135.

[58] Jared T Simpson et al. "ABySS: a parallel assembler for short read sequence data". In: *Genome research* 19.6 (2009), pp. 1117–1123.

[59] Temple F. Smith and Michael S. Waterman. "Identification of Common Molecular Subsequences". In: *Journal of Molecular Biology* 147.1 (), pp. 195–197.

[60] Yatish Turakhia, Gill Bejerano, and William J Dally. "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly". In: *ACM SIGPLAN Notices*. Vol. 53. 2. ACM. 2018, pp. 199–213.

[61] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782. DOI: 10.1145/79173.79181. URL: https://doi.org/10.1145/79173.79181.

[62] Xin Victoria Wang et al. "Estimation of sequencing error rates in short reads". In: *BMC bioinformatics* 13.1 (2012), p. 185.

[63] Sebastian Will. *Lecture notes in Introduction to Computational Molecular Biology, Foundations of Structural Bioinformatics*. Last accessed 05/22/20. 2011. URL: https://math.mit.edu/classes/18.417/Slides/alignment.pdf.

[64] Chuan-Le Xiao et al. "MECAT: fast mapping, error correction, and de novo assembly for single-molecule sequencing reads". In: *Nature Methods* 14.11 (2017), p. 1072.

[65] Katherine Yelick et al. "The parallelism motifs of genomic data analysis". In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190394.

[66] Alberto Zeni et al. "LOGAN: High-Performance GPU-Based X-Drop Long-Read Alignment". In: *arXiv preprint arXiv:2002.05200* (2020).

[67] Qingpeng Zhang et al. "These Are Not the K-mers You Are Looking For: Efficient Online K-mer Counting Using a Probabilistic Data Structure". In: *PLOS ONE* 9.7 (July 2014), pp. 1–13. DOI: 10.1371/journal.pone.0101271. URL: https://doi.org/10.1371/journal.pone.0101271.

[68] Wenyu Zhang et al. "A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies". In: *PloS one* 6.3 (2011), e17915.

[69] Zheng Zhang et al. "A greedy algorithm for aligning DNA sequences". In: *Journal of Computational biology* 7.1-2 (2000), pp. 203–214.

# Appendix A

# Alternative Mathematical Formulations of Long Read Overlap and Alignment

This section provides one of the alternative formulations of overlap hypergraph construction, traversal, and refinement that we considered - many others were considered and are possible. Another is presented in our associated work [23]. The hypergraph abstraction in this work, introduced in Chapter 2, was chosen because it can describe each step in reference to a single abstract object (one hypergraph), rather than two or three, and it is the most general – it can be used to describe any approach to long-read to long-read similarity detection and alignment. Alternatives may be useful for future work. The notation introduced is specific to this appendix and may not be consistent with the main publication.

## A.1 *k-mer* Based Long Read Overlap Detection as Bipartite Graph Construction and Refinement

We can understand the $k$-mer based long read overlap detection problem as the construction and refinement of a bipartite graph. Let $G = \{S, T, E, \psi\}$. The reads are represented by vertices $s \in S$, and the $k$-mers by vertices $t \in T$. The bag of corresponding vertex-weights (read sizes) is $\psi$. Only $S$ is given as input to the problem. $T$, $E$, and $\psi$ must be computed from $S$. The construction process begins by computing $k$-mers from the set of input reads, $S$. An edge $e_{(s,t)} \in E$ is constructed for every read $s \in S$ that contains the corresponding $k$-mer, $t \in T$. We assume there is no isolated vertex in $S$ (every read contains at least one $k$-mer, and typically many $k$-mers, and hence each vertex, $s \in S$, has at least one incident edge). At the end of construction, $G$ is a complete bipartite graph. After $G = \{S, T, E, \psi\}$ is constructed, the first step of refinement is to remove all degree-1 vertices in $T$ ($k$-mers found in exactly 1 read), since these are not useful for analyzing clusters or relationships between vertices (reads). The refinement process then proceeds by removing vertices $t \in T \leftrightarrow degree(t) > m$,

for some given $m$ representing a "noise" threshold. These are $k$-mers so common that they do not distinguish interesting clusters. The result is a rank $m$ bipartite graph.

## A.2 Partitioning the Bipartite Graph

The input is the set of reads, vertices of $S$, and respective vertex-weights, $\psi$, such that $\forall s \in S \exists \psi_s \in \psi$. To distribute the load, we compute partitions of $S$, $P_S$, such that $\forall_{i=1}^{|P_S|} P_i \in P_S$ $(\Sigma_{\forall v \in P_i}\ \psi_v) \approx \frac{\Sigma \psi}{|P_S|}$. For each $k$-mer computed from an input read $s \in S$, we construct a vertex $\{t\} \cup T$, and an edge $\{e_{(s,t)}\} \cup E$. Note, the vertex-weight of each $t \in T$ is uniformly $k$. Therefore, we partition $T$ such that the cardinality of all partitions $p \in P_T$ is roughly equal. We assume no correspondence between the partitioning of $S$, $P_S$, and the partitioning of $T$, $P_T$.

## A.3 Many-to-Many Pairwise Alignment as Hypergraph Construction and Traversal

By traversing the bipartite graph $G$, described in section A.1, the alignment computation hypergraph, $G' = \{V, H, \psi, w\}$, can be constructed. We construct a vertex $\{v_{(a,b)}\} \cup V\ \forall\ a, b \in S \leftrightarrow \exists e_{(a,t)} \wedge e_{(t,b)} \in E$. That is, for each path $(a, t, b) \in G$, the associated pairwise alignment computation will be represented by a vertex $\{v_{(a,b)}\} \cup V, V \in G'$. The hyperedges $H \in G'$ are input reads shared by multiple pairwise alignment computations. The the vertex set $S \in G$ corresponds exactly to the hyperedge set $H \in G'$. Furthermore, for each vertex (computation), $v \in V$, there is an associated weight $w_v$, which represents a variable computational cost, later important for load balancing. The final alignment computation is a traversal of this hypergraph. For each vertex, $v_{(a,c)} \in V$, the pairwise alignment of reads $a, b \in H$ is computed, and $v_{(a,c)}$ is labeled with the corresponding alignment information. For any alignment that does not meet the predetermined alignment score criteria, the corresponding vertex $v_{(a,b)}$ is removed.