Scalable Parallel Algorithms for Genome Analysis

by

Evangelos Georganas

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 A. Yelick, Chair
Professor James W. Demmel
Professor Daniel S. Rokhsar

Summer 2016

Scalable Parallel Algorithms for Genome Analysis

Abstract

Scalable Parallel Algorithms for Genome Analysis

by

Evangelos Georganas

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Katherine A. Yelick, Chair

A critical problem for computational genomics is the problem of *de novo* genome assembly: the development of robust scalable methods for transforming short randomly sampled "shotgun" sequences, namely *reads*, into the contiguous and accurate reconstruction of complex genomes. These reads are significantly shorter (typically hundreds of bases long) than the size of chromosomes and also include errors. While advanced methods exist for assembling the small and haploid genomes of prokaryotes (e.g. cells without nuclei), the genomes of eukaryotes (e.g. cells with nuclei) are more complex. Moreover, de novo assembly has been unable to keep pace with the flood of data, due to the dramatic increases in genome sequencer capabilities, combined with the computational requirements and the algorithmic complexity of assembling large scale genomes and metagenomes.

In this dissertation, we address this challenge head on by developing parallel algorithms for de novo genome assembly with the ambition to scale to massive concurrencies. Our work is based on the Meraculous assembler, a state-of-the-art de novo assembler for short reads developed at the Joint Genome Institute. Meraculous identifies non-erroneous overlapping substrings of length k (k-mers) with high quality extensions and uniquely assembles genome regions into uncontested sequences called *contigs* by constructing and traversing a de Bruijn graph of k-mers, a special type of graph that is used to represent overlaps among k-mers. The original reads are subsequently aligned onto the contigs to obtain information regarding the relative orientation of the contigs. Contigs are then linked together to create *scaffolds*, sequences of contigs that may contain gaps among them. Finally gaps are filled using localized assemblies based on the original reads.

Efficient parallelization of the core algorithms in the Meraculous pipeline exhibit numerous challenges, including the irregular communication patterns. First, we design efficient scalable algorithms for k-mer analysis and contig generation. K-mer analysis is characterized by intensive communication and I/O requirements and our parallel algorithms successfully reduce the memory requirements by 7×. Then, contig generation relies on efficient parallelization of the de Bruijn graph construction and traversal, which necessitates a distributed hash table and is a key component of most de novo assemblers. The underlying de Bruijn graphs are characterized by high diameter and therefore the parallelization of the corresponding algorithms is formidable. We present a novel algorithm that leverages one-sided communication capabilities of the Unified Parallel C (UPC) to facilitate the requisite fine-grained, irregular parallelism and the avoidance of data hazards. The sequence alignment is characterized by intensive I/O and large computation requirements. We introduce merAligner, a highly parallel sequence aligner that implements a seed–and–extend algorithm and employs parallelism in all of its components. We employ a handful of optimizations in order to scale efficiently to massive concurrencies and results show that merAligner outperforms other state-of-the-art tools by orders of magnitude. Finally, this dissertation details the parallelization of the remaining scaffolding modules, enabling the first massively scalable, high quality, complete end-to-end de novo assembly pipeline. Experimental large-scale results on the NERSC Edison Cray XC30 system using human and wheat genomes demonstrate efficient performance and scalability on thousands of cores. Compared to the original Meraculous code, which has limited scalability, and requires approximately 48 hours to assemble the human genome, our pipeline called HipMer computes the assembly in only 4 minutes using 23,040 cores of Edison – an overall speedup of approximately 720×.

In the last part of the dissertation we tackle the problem of metagenome assembly. Metagenomics is currently the leading technology to study the uncultured microbial diversity and delineating the microbiome structure and function. While accessing an unprecedented number of environmental samples that consist of thousands of individual microbial genomes is now possible, the bottleneck is becoming computational, since the sequencing cost improvements notoriously exceed that of Moore's Law. Assemblies of metagenomes into long contiguous sequences are not only critical for the identification of the usually long biosynthetic clusters but are also key for enabling the discovery of new lineages of life and viruses. Metagenome assembly is further complicated (compared to single genome assembly) by repeated sequences across genomes, polymorphisms within a species and variable frequency of the genomes within the sample. In our work we repurpose HipMer components for the problem of metagenome assembly and we design a versatile, high-performance metagenome assembly pipeline that outperforms state-of-the-art tools in both quality and performance.

To my parents and my brother.

Contents

Acknowledgments

First and foremost, I am grateful to my advisor and mentor, Kathy Yelick, for all her support over the last five years. She gave me the chance to be part of UC Berkeley and work with her and so many talented colleagues on interesting problems. From the first time I worked with her, she has been always there to provide her advice and showed me how to conduct research that matters. I would also like to thank the rest of my thesis committee, Jim Demmel, Dan Rokhsar and Satish Rao for their valuable feedback on this work.

I would like to especially acknowledge the collaborators in the HipMer project, which constitutes the main part of this dissertation: Aydın Buluç, Steve Hofmeyr, Rob Egan, Lenny Oliker, Jarrod Chapman and Dan Rokhsar. Without their help and advice, this work would never be possible. In particular I would like to thank Jarrod Chapman and Dan Rokhsar for introducing me to the problem of genome assembly and guiding me through the project.

Additionally, I would like to thank a number of other collaborators with whom I had the privilege to work in various research projects: Penporn Koanantakool, Michael Driscoll, Edgar Solomonik, Jorge González-Domínguez and Yili Zheng. Many thanks to all the members of the BEBOP group, the ParLab/ASPIRE lab fellow students and the UPC group at LBNL for the numerous interesting research discussions. Many thanks also to the administrative staff at UC Berkeley and LBNL for their invaluable help.

I would like to thank my friends from Greece and those here at Berkeley for all the unforgettable moments we have had together; they have made and keep making my life exciting. Thanks to Mahi for the wonderful time we are having together. Finally, I would like to express my deep gratitude to my mother Aliki, my father Alexandros and my brother Thodoris, for their unconditional love and support all these years. This love shaped me into the person I am today and without their support I could not have done anything.

This work was supported in part by the Department of Energy, Office of Science, Advanced Scientific Computing Research X-Stack Program under Lawrence Berkeley National Laboratory Contract DE-AC02-05CH11231 and under UC Berkeley Contract No. DOE DE-SC0008700. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Chapter 1

Introduction

Genomes are the fundamental biochemical elements underlying inheritance, represented by chemical sequences of the four DNA "letters" A, C, G, and T that stand for the nucleobases Adenine, Cytosine, Guanine and Thymine respectively. Genomes encode the basic software of an organism, defining the proteins that each cell can make, and the regulatory information that determines the conditions under which each protein is produced, allowing different organs and tissues to establish their distinct identities and maintain the stable existence of multicellular organisms like humans. Sequences that differ by as little as one letter can cause the expression of proteins that are defective or are inappropriately expressed at the wrong time or place. These differences underlie many inherited diseases and disease susceptibility.

Each organism's genome is a specific sequence, ranging in length from a few million letters for a typical bacterium to 3.2 billion letters for a human chromosome to over 20 billion letters for some plant genomes, including conifers and bread wheat. Genomes differ between species and between individuals within species; for example, two healthy human genomes typically differ at more than three million positions, and each can contain over ten million letters that are absent in the other. Genomes mutate between every generation and even within individuals as they grow, and some of those mutations can drive cells to proliferate and migrate inappropriately, leading to diseases such as cancer. We do not yet know which sequence differences are important but hope to learn these rules by sequencing millions of healthy and sick people, and comparing their genomes.

Determining a genome sequence "de novo" (that is, without reference to a previously determined sequence for a species) is a challenging computational problem. Modern sequencing instruments can cost-effectively produce only short, *erroneous* sequence fragments of 100-250 letters, read at random from a genome (so-called "shotgun" sequencing). A billion such short reads can be produced for around $1,000, enough to redundantly sample the human genome thirty times in overlapping short fragments. The computational challenge of *genome assembly* is then to reconstruct chromosome sequences from billions of overlapping short, erroneous sequence fragments, bridging a six order of magnitude gap between the length of the individual raw sequence reads and a complete chromosome.

Reconstructing a long sequence from short substrings is in general an NP-hard problem, and must rely on heuristics and/or take advantage of specific features of genome sequences [98, 56]. Current genome assembly pipelines typically rely on single node, large memory (e.g. 1 TB) architectures, and can take a week to assemble a single human genome, or even several months for larger genomes like loblolly pine [110]. These approaches cannot be used in clinical settings or on genomes that exceed the memory footprint of Terabytes. While some distributed memory parallel algorithms have been developed [97, 9, 79, 51], they do not scale to massive concurrencies as they exhibit algorithmic bottlenecks and the irregular access patterns that are inherent to these algorithms amplify the distributed memory parallelization overheads.

To add more urgency to the problem, over the last few years increasing attention has been devoted to the microbiome, the collective of single-celled organisms that live in diverse environments, including the human gut and other cavities and surfaces on the body. The human microbiome contains several times as many cells as are in the human body, and the microbes that live in and on us are increasingly linked to health [2]. But most of these microbes are undescribed and uncultivated, and their role in health or disease unknown. An increasingly important way of discovering and characterizing these unknown microbes is direct shotgun sequencing of DNA from an environment, such as the human gut, producing a "metagenome". If the genomes of the constituent microbes can be assembled from the resulting set of short fragments, we can gain valuable information about the functional and role of these organisms. The number of publicly available metagenome datasets is rapidly increasing where in the span of just two years (2011-2013), the number of metagenome datasets nearly quadrupled and the number of identified protein coding genes increased by 119$\times$ [76], far exceeding Moore's law of computing. An added complexity of metagenomes assembly is that in a given microbiome some species are common and others rare; each organism contributes to the raw data in proportion to its prevalence and hundreds or thousands of species can be present.

This dissertation addresses the aforementioned challenges by developing parallel algorithms for de novo assembly with the ambition to scale to massive concurrencies. The result of this work is HipMer [33], an end-to-end high performance de novo assembler designed to scale to massive concurrencies. HipMer uses (i) high performance computing clusters or supercomputers for both memory size and speed, (ii) a global address space programming model via Unified Parallel C (UPC) [37] to permit random accesses across the aggregate machine memory, and (iii) parallel graph algorithms and hash tables, optimized for the statistical characteristics of the assembly process to reduce communication costs and increase parallelism. Our work is based on the Meraculous [19, 18] assembler, a state-of-the-art de novo assembler for short reads developed at the Joint Genome Institute [54]. Meraculous is a hybrid assembler that combines aspects of de Bruijn-graph-based assembly with overlap-layout-consensus approaches and is ranked at or near the top in most metrics of the Assemblathon II competition [13]. Meraculous leverages base quality scores from sequencing instruments to identify non-erroneous overlapping substrings of length k (k-mers) with high quality extensions and then uniquely assembles genome regions into uncontested sequences called *contigs*. The original reads are subsequently aligned onto the contigs to obtain information regarding the relative orientation of the contigs. The last stage of the Meraculous pipeline is called *scaffolding* where the read-to-contigs alignments are assessed to stitch together multiple contigs into longer scaffolds that may contain gaps. Finally gaps are filled using localized assemblies based on the original reads; these localized assemblies involve only localized reads and therefore are able to extract information that is hard to identify by examining the complex, entire read dataset. The original Meraculous used a combination of serial, shared memory parallel, and distributed memory parallel code. The size and complexity of genomes that could be assembled with Meraculous was limited by both speed and memory size. Our goal was a fast, scalable parallel implementation that could use the combined memory of a large scale parallel machine and our work [35, 34, 33] has covered all aspects of the single genome assembly pipeline: k-mer analysis, contig generation, sequence alignment, scaffolding and gap closing.

The first step of the pipeline, namely k-mer analysis, is to process the erroneous input reads and extract error-free k-mers. The reads are chopped into overlapping k-mers that start at every position in the input (i.e. a read of length L will have $L - k + 1$ k-mers) and a count is kept for each k-mer occurring more than once. The generated k-mers are preprocessed and only those who appear more times than a user-specified threshold are kept. Also, we further filter the survived k-mers and maintain only those with unique high-quality forward and backward single-base extensions; we consider that a k-mer extension is of "high quality" if it appears to extend that k-mer more times than a user-specified threshold. These remaining k-mers are considered to be error-free. One of the difficulties with performing k-mer analysis in distributed memory is that the size of the intermediate data (the set of k-mers) is significantly larger than the input, since each read is subsequenced with overlaps of $k - 1$ base pairs. Also, the massive input read datasets stress the I/O system while the k-mer counting phase requires high communication bandwidth. Finally, high frequency k-mers constitute a source of load imbalance and pose a scalability impediment at large concurrencies. Our work overcomes all these shortcomings by employing novel algorithmic and optimization techniques.

The output of the k-mer analysis is a set of k-mers with high quality extensions. These k-mers are used to form a de Bruijn graph, where the k-mers are the vertices in the graph and two k-mers are connected with an edge if they overlap by $k - 1$ consecutive bases. We store this de Bruijn graph in a compact way using a hash table: the keys in the hash table represent the vertices and the values represent the adjacent edges of the corresponding vertices, i.e. the high-quality extensions of the k-mers. Then, by traversing the de Bruijn graph we find the connected components which constitute the contigs. However, parallelizing the contig generation exhibits a few challenges. The first challenge involves the size of the de Bruijn graph which can be very large depending on the genome size. For complex eukaryotes the corresponding hash table requires hundreds of gigabytes to tens of terabytes of memory. Also, the underlying graph of k-mers is characterized by high-diameter where parallel Breadth First Search (BFS) approaches do not scale well and this fact poses a second challenge. In order to deal with the first challenge, we distribute the hash table to multiple processors by leveraging the global address space of UPC which obviates the need for large shared memory hardware. However, we introduce communication and synchronization overheads in the parallel graph construction. We further optimize this process by applying dynamic message aggregation and we minimize the number of messages and the number of required synchronization events. In regard to the de Bruijn graph traversal, we develop a novel parallel algorithm that employs massive concurrency. This algorithm exploits the one-sided communication capabilities of UPC, remote atomics and a lightweight synchronization protocol. Finally, we identify sources of locality in this kind of graph and design an appropriate graph partitioning scheme. These methods can significantly reduce the volume of communication during the graph traversal, yielding a communication-avoiding parallel algorithm. The high-performance contig generation has been used for the whole-genome shotgun assembly of the highly repetitive 16 Gbp (Giga base pairs) hexaploid bread wheat genome [20]. The contig generation of the wheat genome required 73 seconds on 3,840 cores of the Edison supercomputer [26], where in aggregate we had available 10 Tbytes of memory.

The contig generation produces highly confident regions of contiguous sequences, but they can be further extended by leveraging information from the original reads. The foremost task in the contigs's extension process is to align the input reads onto the contigs. The sequence alignment algorithm in Meraculous uses a *seed-and-extend* paradigm where: (1) the contig sequences are decomposed into overlapping seeds of length k which are subsequently indexed with a hash table (keys are the seeds and values are the corresponding contigs they were extracted from) and (2) we extract seeds from the reads, look them up in the index data structure, locate candidate contigs for alignment and eventually perform an extension algorithm with the read and the candidate contig as inputs. There are a few parallel sequence alignment tools but they all suffer mainly from two limitations. First, they build the seed index serially and then replicate it to the available processors. However, the seed index construction requires a lot of processing time for large genomes and the serial procedure imposes a serialization bottleneck on the critical path of the parallel pipeline. Second, the seed index for large eukaryote genomes requires hundreds of gigabytes of memory and thus can not fit even in a typical modern supercomputer node. To overcome these major scalability impediments, we have developed merAligner [34], a fully parallel sequence aligner that employs parallelism in all of its components. MerAligner distributes the hash table that represents the seed index to multiple nodes. Also, merAligner leverages dynamic message aggregation at the construction of the distributed hash table and software caching schemes to reduce the communication volume during the aligning phase. Additionally, merAligner preprocesses the contig sequences to extract properties enabling exact sequence matching with minimal communication.

Regarding the remaining steps in the scaffolding module, we assess the read-to-contig alignments from the previous alignment step and generate *links* among contigs. In order to assess these links in parallel, we utilize again distributed hash tables with communication optimizations. Finally, we form a graph of contigs using the links among them and by identifying the connected components in this graph we form *scaffolds* of contigs. Each scaffold may contain gaps between the contigs, which are efficiently closed by employing a mini-assembly procedure. The parallelization of the scaffolding modules enables the first massively scalable, high quality, complete end-to-end de novo assembly pipeline, which we call HipMer. Experimental large-scale results on the NERSC Edison Cray XC30 using human and the wheat genomes demonstrate efficient performance and scalability on thousands of cores. Compared with the original Meraculous code, which has limited scalability, and requires approximately 48 hours to assemble the human genome, HipMer employs an efficient UPC implementation, computing the assembly in only 4 minutes using 23,040 cores of Edison – an overall speedup of approximately 720×.

To tackle the metagenome assembly problem, we repurpose the high performance algorithmic modules in HipMer and implement a metagenome assembly metagenome pipeline called *meta-HipMer*. The metagenome datasets are in general much larger than the single genome case in order to resolve all of the individual genomes; this affects both the input size and the memory required for hash tables of k-mers. The meta-HipMer algorithm consists of two main components: (1) the iterative contig generation and (2) the scaffolding component. The iterative contig generation aims to eliminate the quality trade-off that different k-mer sizes induce in de Bruijn graph based assemblers. Typically, a small value of k is appropriate for low-coverage genome areas since it allows sufficient number of overlapping k-mers to be found and as a result the underlying sequences can be assembled to longer contigs. On the other hand, a large value of k is better suited for the high-coverage regions since a sufficient number of overlapping, long k-mers can be found and repetitive regions are disambiguated by such long k-mers. In each iteration of the contig generation step, meta-HipMer applies a handful of transformations on the de Bruijn graphs, such as "bubble" merging due to to Single Nucleotide Polymorphisms (SNPs) and iterative graph pruning aiming to eliminate erroneous graph branches. To accommodate low coverage genomes, meta-HipMer leverages a localized assembly algorithm that is able to extract critical information from the entire dataset. Eventually, the HipMer's scaffolding module is utilized and adapted to finalize the metagenome assembly. Large scale experimental results with the meta-HipMer pipeline indicate that our approach outperforms state-of-the-art tools in both quality and performance.

In the last part of this dissertation we present an architectural analysis of core parallel operations encountered in our assembly pipelines. First, we investigate the communication aspects that are stressed during the pipeline and we illustrate the impact of the communication infrastructure on the parallel contig generation and sequence alignment. Then, we develop micro-benchmarks to assess the behavior of such irregular access patterns at different scales and we compare the performance of the micro-benchmarks with the empirical performance of the pipeline's modules. These micro-benchmarks provide naturally a roofline model [103] and we demonstrate that our parallel implementations are efficient. Finally, we highlight the necessity for scalable, parallel I/O infrastructure in modern supercomputers in order to accommodate genome assembly pipelines that deal with massive datasets. The results of our architectural analysis indicate bottlenecks in current architectures and provide insights on how to improve the performance of such patterns in future systems.

1.1 Contributions

The main contributions of this dissertation include new parallel algorithms designed for global address space programming, i.e. a shared memory style algorithms with reads, writes and atomic updates to remote data, but having data layout and communication optimizations for distributed memory scaling:

- A global address space algorithm for de Bruijn graph construction that represents the graph as a hash table.
- A global address space graph traversal algorithm that is suitable for high-diameter graphs where traditional BFS approaches fail to scale.
- A new parallel sequence alignment algorithm that employs various communication and computation optimizations.
- A new parallel scaffolding algorithm for a global address space with various communication optimizations.
- A new parallel algorithm for metagenome assembly that efficiently deals with the idiosyncrasies of metagenome datasets.

In addition, the thesis describes novel scalable implementations using the Unified Parallel C (UPC) language for partitioned global address space programming:

- A scalable implementation of the graph construction algorithm with asynchronous communication allowing processors to perform communication on demand, i.e. at different rates based on their own distribution of keys, and dynamic message aggregation to amortize communication overhead.
- A fast implementation of graph traversal with a remote synchronization approach that resolves data races while minimizing synchronization overhead.
- A locality aware hashing strategy that reduces communication in the de Bruijn graph traversal for genomes that are similar to previously assembled ones as is the case for human assembly.
- A scalable sequence alignment implementation that distributes the index of the "reference" genome as a distributed hash table, deploys software caching to reduce communication and has a preprocessing optimization to quickly identify exact matches, avoiding both communication and computation.
- A complete parallel implementation of single genome assembly algorithm, HipMer, which matches the quality of the original Meraculous code.

- A distributed memory parallel metagenome assembly pipeline, meta-HipMer, which produces high-quality results, meeting or improving on other state-of-the-art metagenome assemblers, and scales to much larger data sets.

Finally, we analyze the performance of our codes, noting that the genome assembly workload is unlike most high performance simulation codes:

- Experimental analysis of the entire genome assembly pipeline on a Cray XC30 supercomputer using the human genome and the grand-challenge wheat genome datasets, showing unprecedented performance and scalability, attaining an overall runtime of 3.91 minutes for the human DNA at 23K cores on the Cray XC30 while the original pipeline required 48 hours on a large shared memory machine.

- An architectural analysis of key parallel operations encountered in our assembly pipelines, indicating bottlenecks in current architectures and providing insights on how to improve the performance of such parallel operation in future systems.

The rest of this dissertation is organized as follows. Chapter 2 describes the fundamental concepts that we build upon in this dissertation. Chapters 3 and 4 describe the parallel algorithms for k-mer analysis and contig generation respectively. Chapter 5 details that parallel sequence alignment algorithm while Chapter 6 presents the parallelization of the scaffolding and gap closing modules. Chapter 7 provides a quality assessment of the HipMer single genome assembly pipeline, illustrates its end-to-end scalability results and compares its performance to other assemblers. Chapter 8 introduces the parallel algorithms in the meta-HipMer metagenome assembly pipeline along with experimental results. Chapter 9 illustrates an architectural analysis of core parallel operations in the HipMer and meta-HipMer pipeline. Finally, Chapter 10 presents related work and Chapter 11 concludes this dissertation.

Chapter 2

Background

In this chapter, we describe the fundamental concepts that we build upon in this dissertation. We start by reviewing the Meraculous [19, 18] single genome assembly pipeline and its main algorithmic components. This pipeline is the basis for our parallel algorithms. We then provide a brief overview of the Partitioned Global Address Space Model (PGAS) memory model in Unified Parallel C (UPC) [37] which lies at the heart of our parallelization techniques. Finally, we discuss aspects of distributed hash tables which constitute our main distributed data structure. In particular, we examine use cases of distributed hash tables that enable various optimizations in a PGAS model. These use cases will be used as point of reference throughout this dissertation.

2.1 The Meraculous Assembly Pipeline

The input to a genome assembly pipeline is a set of short, erroneous sequence fragments of 100-250 letters, read at random from a genome (see Figure 2.1). Note that the genome is redundantly sampled at a depth of coverage d. Typically these reads fragments come in

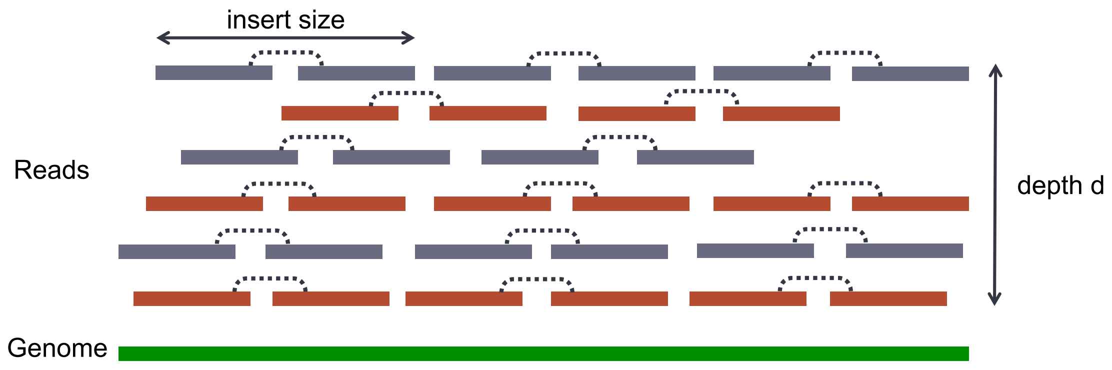
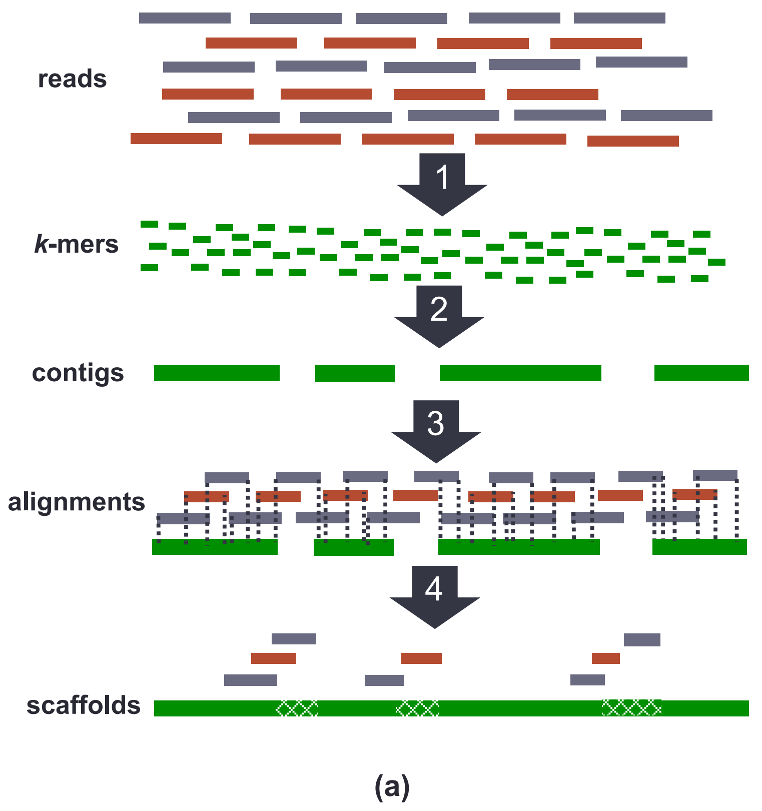


Figure 2.1: Reads extracted from a genome with a depth of coverage d.



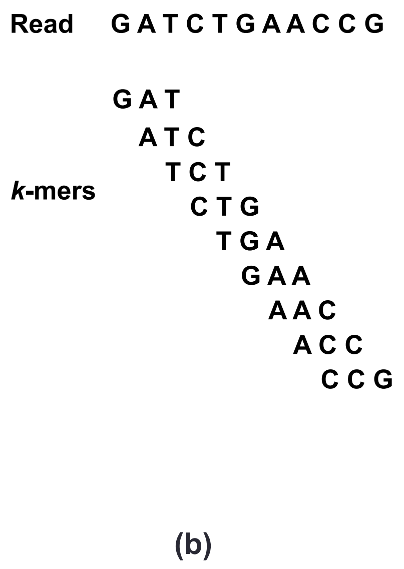


Figure 2.2: (a) The Meraculous assembly pipeline. (b) Extracting k-mers ($k = 3$) from the read `GATCTGAACCG`.

pairs and this information will be further exploited in the pipeline. Paired reads are also characterized by the *insert size*, the distance between the two ends of the reads. Thus, given the read lengths and the corresponding insert size, we have an estimate for the gap between the paired reads. Typically the reads are grouped into libraries and each library is characterized by a nominal insert size and its standard deviation. Libraries with different insert sizes play a significant role in the assembly process, as will be explained later in this section.

The Meraculous pipeline consists of four major stages (see Figure 2.2(a)):

1. **K-mer analysis:** The input reads are processed to exclude errors. First, the reads are chopped into k-*mers*, which are overlapping sequences of length k. Figure 2.2(b) shows the k-mers (with $k = 3$) that are extracted from a read. Then, the k-mers extracted from all the reads are counted and those that appear fewer times than a threshold are treated as erroneous and discarded. Additionally, for each k-mer we keep track of the two neighboring bases in the original read it was extracted from (henceforth we call these bases *extensions*). The result of k-mer analysis is a set of k-mers and their corresponding extensions that with high probability include no errors.

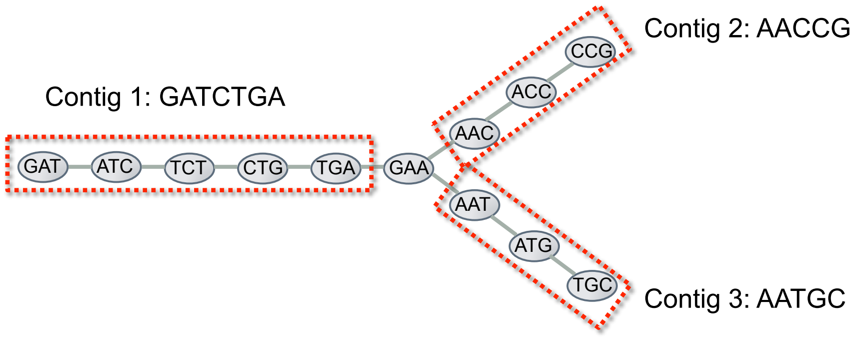


Figure 2.3: A de Bruijn graph of k-mers with $k = 3$.

It is worth noting that the redundancy d in the read data set is crucial in the process of excluding the errors implicitly. More specifically, an error at a specific read location yields k erroneous k-mers[1]. However, there are more reads covering the same genome location due to the redundancy d. More precisely, given the read length L we expect to find a k-mer on average $f = d \cdot (1 - (k-1)/L)$ times in the read data set where f is the mean of the Poisson distribution of key-frequencies [68] and most of these k-mer occurrences will be error-free. Therefore, if we find a particular k-mer just one or two times in our read dataset, then we consider that to be erroneous. On the other hand, k-mers that appear a number of times proportional to d are likely error-free.

2. **Contig generation:** The resulting k-mers from the previous step are stored in a de Bruijn graph. This is a special type of graph that represents overlaps in sequences. In this context, k-mers are the vertices in the graph, and two k-mers that overlap by $k-1$ consecutive bases whose corresponding extensions are compatible are connected with an undirected edge in the graph (see Figure 2.3 for a de Bruijn graph example with $k = 3$).

   Due to the nature of DNA, the de Bruijn graph is extremely sparse. For example, the human genome's adjacency matrix that represents the de Bruijn graph is a $3 \cdot 10^9 \times 3 \cdot 10^9$ matrix with between two and eight non-zeros per row for each of the possible extensions. In Meraculous only k-mers which have unique extensions in both directions are considered, thus each row has exactly two non-zeros.

   Using a direct index for the k-mers is not practical for realistic values of k, since there are 4^k different k-mers. A compact representation can be leveraged via a hash table: A vertex (k-mer) is a key in the hash table and the incident vertices are stored implicitly as a two-letter code `[ACGT][ACGT]` that indicates the unique bases that immediately precede and follow the k-mer in the read dataset. By combining the key and the two-letter code, the neighboring vertices in the graph can be identified.

[1]If the error does not occur at the k first/last bases of the read, then k erroneous k-mers are generated. Otherwise fewer erroneous k-mers are extracted.

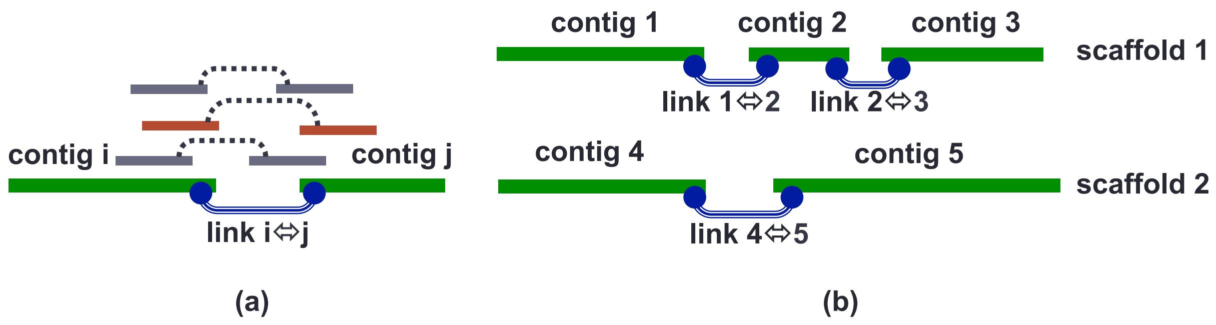


Figure 2.4: (a) A link between contigs i and j that is supported by three read pairs. (b) Two scaffolds formed by traversing a graph of contigs.

In Figure 2.3 all k-mers (vertices) have unique extensions (neighbors) except from the vertex `GAA` that has two "forward neighbors", vertices `AAC` and `AAT`. From the previous k-mer analysis results we can identify the vertices that do not have unique neighbors. In the contig generation step we exclude from the graph all the vertices with non-unique neighbors. Via construction and traversal of the underlying de Bruijn graph of k-mers the connected components in the graph are identified, which are linear chains of k-mers, called *contigs*. The connected components have linear structure since we exclude from the graph all the "fork" nodes or equivalently the k-mers with non-unique neighbors. The contigs are (with high probability) error-free sequences that are typically longer than the original reads. In Figure 2.3 by excluding the vertex `GAA` that doesn't have unique neighbors, we find three linear connected components that correspond to three contigs.

3. **Aligning reads onto contigs:** In this step we map the original reads onto the generated contigs. This mapping provides information about the relative ordering and orientation of the contigs and will be used in the final step of the assembly pipeline.

   The Meraculous pipeline adopts a seed-and-extend algorithm in order to map the reads onto the contigs. First, the contig sequences are indexed by constructing a seed index, where the seeds are all substrings of length k that are extracted from the contigs. This seed index is then used to locate candidate read-to-contig alignments. Given a read, we extract seeds of length k, look them up in the seed index and as a result we get candidate contigs that are aligned with the read because they share common seeds. Finally, an extension algorithm (e.g. Smith-Waterman [99]) is applied to extend each found seed and local alignments are returned as the final result.

4. **Scaffolding and gap closing:** The scaffolding step aims to "stitch" together contigs and form sequences of contigs called *scaffolds* by assessing the paired-end information from the reads and the reads-to-contigs alignments. Figure 2.4(a) shows three pairs of reads that map onto the same pair of contigs i and j. Hence, we can generate a link

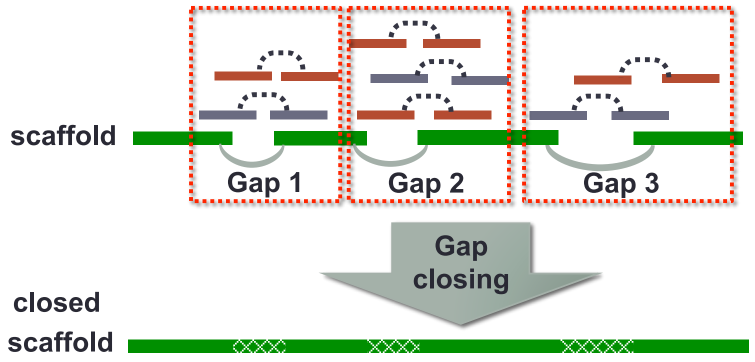


Figure 2.5: The gap closing procedure.

that connects contigs i and j. By generating links for all the contigs that are supported by pairs of reads we create a graph of contigs (see Figure 2.4(b)). By traversing this graph of contigs we can form chains of contigs which constitute the scaffolds. Note that libraries with large insert sizes can be used to generate long-range links among contigs. Additionally, scaffolding can be performed in an iterative way by using links generated from different libraries at each iteration.

After the scaffold generation step, it is possible that there are gaps between pairs of contigs. We then further assess the reads-to-contigs mappings and locate the reads that are placed into these gaps (see Figure 2.5). Ultimately, we leverage this information and close the contig gaps by performing a mini-assembly algorithm involving only the localized reads for each gap. The outcome of this step constitutes the result of the Meraculous assembly pipeline.

In Chapters 3, 4, 5, 6 and 7 we examine in more detail the algorithms involved in the Meraculous pipeline along with their parallelization.

2.2 The Partitioned Global Address Space Model in Unified Parallel C

The Partitioned Global Address Space (PGAS) model is a communication mechanism employed in parallel programming languages. In this model, any thread is allowed to directly access memory on other threads. In the PGAS model, two threads may share the same physical address space or they may own distinct physical address spaces. In the former case, remote-thread accesses can be done directly using load and store instructions while in the latter case a remote access must be translated into a communication event, typically using a communication library such as GASNet [11] or hardware specific layers such as Cray's DMAPP [15] or IBM's PAMI [58].

An alternative communication mechanism typically employed in parallel programming languages is message passing, where the communication is done by exchanging messages between threads (e.g. see the Message Passing Interface (MPI) [40]). In such a communication model, both the sender and the receiver should explicitly participate in the communication event and therefore requires coordinating communication peers to avoid deadlocks. The programmer's burden in such a two-sided communication model can be further exaggerated in situations where the communication patterns are highly irregular as in distributed hash table construction. On the other hand, the PGAS model requires the explicit participation only of the peer that initiates the communication and as a result parallel programs with irregular accesses are easier to implement. Such a communication mechanism is typically referred to as *one-sided communication*. In addition to PGAS languages like Unified Parallel C (UPC) [37] there are programming libraries such as SHMEM [17] and MPI 3.0 [24] with one-sided communication features. In Chapter 9 we describe some of the specific one-sided capabilities that are required by our parallel algorithms.

Unified Parallel C (UPC) is an extension of the C programming language designed for high performance computing on large-scale parallel machines by leveraging a PGAS communication model. UPC utilizes a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time. On top of its one-sided communication capabilities, UPC provides global atomics, locks and collectives that facilitate the implementation of synchronization protocols and common communication patterns. In short, UPC combines the programmability advantages of the shared-memory programming paradigm and the control over data layout and performance of the message passing programming paradigm. According to the memory model of UPC each thread has a portion of shared and private address space. Variables that reside in the shared space can be directly accessed by any other thread and typically synchronization is required in order to avoid race conditions. On the other hand, variables that live in the private space can be read and written only by the thread owning that particular private address space.

2.3 Distributed Hash Tables in a PGAS Model

A common data structure utilized in subsequent parallel algorithms is the distributed hash table. There is a wide body of work on concurrent hash tables [96, 45, 47, 28, 59, 81, 95] that focuses on shared memory architectures. There is also a lot of work on distributed hash tables (see [6], [90] and survey of Zhang et al. [107]) specially designed for large-scale distributed environments that support primitive `put` and `get` operations. Such implementations do not target dedicated HPC environments and therefore have to deal with faults, malicious participants and system instabilities. Such distributed hash tables are optimized for execution on data centers rather than HPC systems with low-latency and high-throughput interconnects. There are some simple distributed memory implementations of hash tables in MPI [36] and UPC [77], but they are used mainly for benchmarking purposes of the underlying runtime and do not optimize the various operations depending on the use case of the

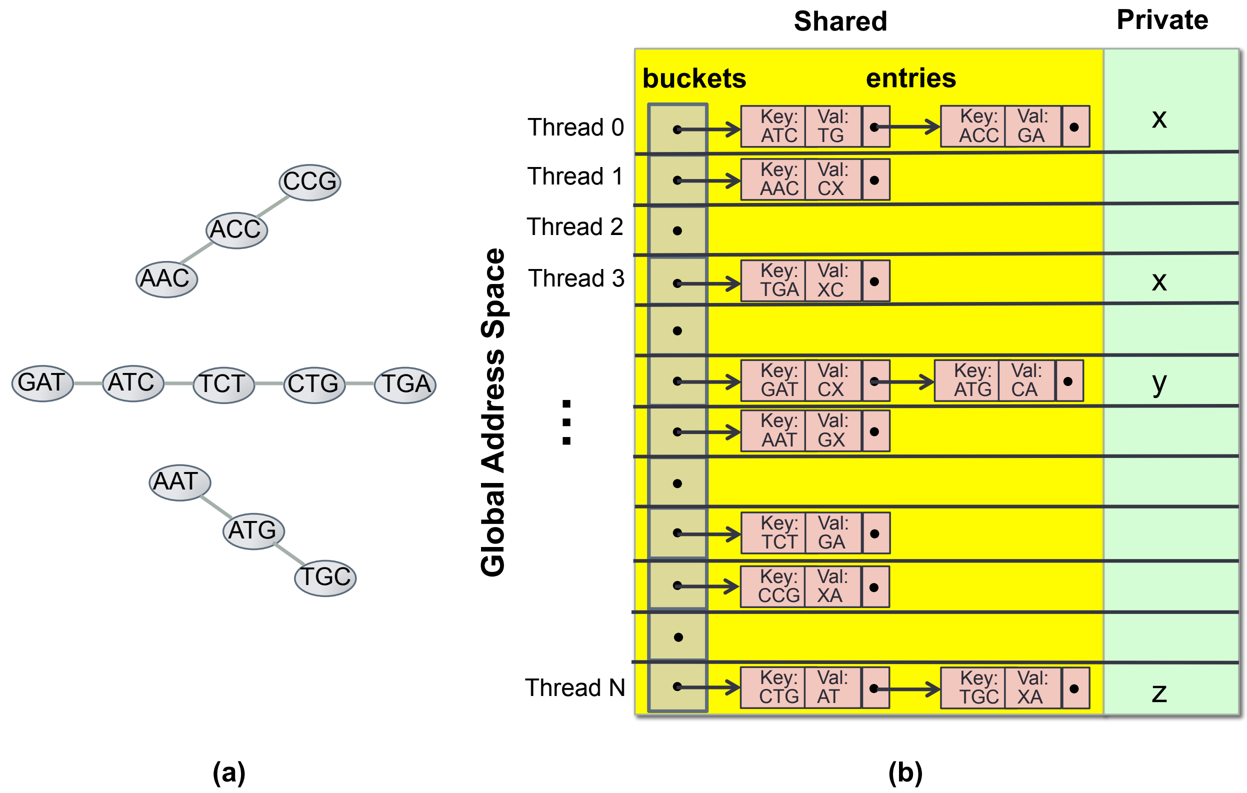


Figure 2.6: (a) A de Bruijn graph of k-mers ($k = 3$). (b) A distributed hash table that represents de Bruijn graph at left.

hash table. In this section we describe the basic implementation of a distributed hash table in a PGAS model. We also identify a handful of use cases for distributed hash tables that enable numerous optimizations for HPC environments.

2.3.1 Basic implementation of a distributed hash table

We will present the vanilla implementation of a distributed hash table by following an example of a distributed de Bruijn graph. Figure 2.6 (a) shows a de Bruijn graph of k-mers with $k = 3$ and Figure 2.6 (b) illustrates its representation in a distributed hash table. A vertex (k-mer) in the graph is a key in the hash table and the incident vertices are stored implicitly as a two-letter code `[ACGTX][ACGTX]` that indicates the unique bases that follow and precede that k-mer. This two letter code is the value member in a hash table entry. Note that the character `X` indicates that there is no neighboring vertex in that direction. By combining the key and the two-letter code, the neighboring vertices in the graph can be identified. More specifically, by concatenating the last $k-1$ letters of a key and the first letter of the value, we get the "forward" neighboring vertex. Similarly, by concatenating the second letter of the value and the first $k-1$ letters of that key, we get the "preceding" neighboring vertex.

In our example, all the hash table entries are stored in the shared address space and thus they can be accessed by any thread. The buckets are distributed to the available threads in a cyclic fashion to achieve load balance. Note that this hash table implementation utilizes a chaining rule to resolve collisions in the buckets (entries with the same hash value). We emphasize here that the hash tables involved in our algorithms can be gigantic (hundreds of Gbytes up to tens of Tbytes) and cannot fit in a typical shared-memory node. Therefore it is crucial to distribute the hash table buckets over multiple nodes and in this quest the global address space of UPC is convenient.

In the following subsection we list different use case scenarios of distributed hash tables and in the subsequent chapters we examine efficient algorithms to construct and utilize the hash tables under various conditions.

2.3.2 Use cases of distributed hash tables

Here we identify a handful of use cases for the distributed hash tables that allow specific optimizations in their implementation. These use-cases will be used as points of reference in the chapters that detail our parallel algorithms.

- **Use case 1 – Global Update-Only phase (GUO)**
  The operations performed in the distributed hash table are only global updates with commutative properties (e.g. inserts only). The global hash table will have the same state (although possibly different underlying representation due to chaining) regardless of insert order. The global update-only phase can be optimized by dynamically aggregating fine-grained updates (e.g. inserts) into batch updates. In this way we can reduce the number of messages and synchronization events. We can also overlap computation/communication or pipeline communication events to further hide the communication overhead.

  A typical example of such a use case is a producer/consumer setting where the producers operate in a distinct phase from consumers, e.g. all consumers insert items in a hash table before anything is consumed/read.

- **Use case 2 – Global Reads & Writes phase (GRW)**
  The operations performed during this phase are global reads and writes over the *already inserted entries*. Typically we can't batch reads and/or writes since there might be race conditions that affect the control flow of the governing parallel algorithm. However, we can use global atomics (e.g. compare-and-swap) instead of fine-grained locking in order to ensure atomicity. The global atomics might employ hardware support depending on the platform and the corresponding runtime implementation. We can also build synchronization protocols at a higher level that do not involve the hash table directly but instead are triggered by the results of the atomic operations. Finally, we can implement the delete operation of entries with atomics and avoid locking schemes.

For example consider the consumers in a producer/consumer scenario that compete for the entries of the hash table. The entries may have utilization signatures (i.e. "used" binary flags) that can be accessed via global atomics and indicate that if the corresponding entries have been consumed or not. An orthogonal optimization for this use case scenario is to adopt locality sensitive hashing schemes to increase locality and decrease communication volume/latency overhead of global atomics.

- **Use case 3 – Global Read-Only phase (GRO)**
  In such a use case, the entries of the distributed hash table are read-only and a degree of data reuse is expected. The optimization that can be readily employed is to design software caching schemes to take advantage of data reuse and minimize communication. These caching frameworks can be viewed as "on demand" copying of remote parts of the hash table. Note that the read-only phase guarantees that we do not need to provision for consistency across the software caches. Such caching optimizations can be used in conjunction with locality-aware partitioning to increase effectiveness of the expected data reuse. Initially even if the data is remote, it is likely to be reused later locally.

  A typical example of this use case is a lookup only hash table that implements a database/index. This is a special case of the consumer side in a producer/ consumer setting where the entries can be consumed an infinite number of times.

- **Use case 4 – Local Reads & Writes phase (LRW)**
  In this use case, the entries in the hash table will be further read/written only by the processor owning them. The optimization strategy we employ in such a setting is to use a deterministic hashing from the sender side and local hash tables on the receiver side. The local hash tables ensure that we avoid runtime overheads and also high-performance, serial hash table implementations can be seamlessly incorporated into parallel algorithms.

  For example, consider items that are initially scattered throughout the processors and we want to send occurrences of the same item to a particular processor for further processing (e.g. consider a "word-count" type of task). Each processor can insert the received items into a local hash table and further read/write the local entries from there.

We emphasize that this is not an exhaustive list of use cases for distributed hash tables. Nevertheless, it captures the majority of the computational patterns we identified in our parallel algorithms that will be detailed in the following chapters. Table 2.1 summarizes the optimizations we can employ for the various use cases of the distributed hash tables. Multiple of the aforementioned use cases can be encountered during the lifetime of a distributed hash table; in most of the cases the optimizations can be easily composed (e.g. by having semantic barriers to signal the temporal boundaries of the phases). For example, the Global Update-Only phase can be followed by a Global Read-Only phase in a scenario where a database is

Use case	Dynamic message aggregation	Remote global atomics	Caching of remote entries	Locality sensitive hashing	Serial hash table library
GUO	✓	✓		✓	
GRW		✓		✓	
GRO			✓	✓	
LRW	✓			✓	✓

Table 2.1: Distributed hash table optimizations for various use case scenarios.

first built via insertion of the corresponding items into a hash table and later the distributed data structure is reused as a global lookup table.

Chapter 3

Parallel Algorithms for k-mer Analysis

The parallel algorithms and implementations presented in this Chapter are done primarily by Aydın Buluç and were part of the co-authored papers "Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly" [35] and "HipMer: An Extreme-Scale De Novo Genome Assembler" [33]. We include this material as necessary background for later chapters. In particular, we describe parallel algorithms for the first stage of the assembly pipeline, namely k-mer analysis. First, we present the basic parallel algorithm and then we describe the optimization techniques we employed in order to achieve scalability. Finally, we present large-scale experimental results and conclude the parallel k-mer analysis algorithms.

3.1 Parallel Three-pass k-mer Analysis

K-mers are contiguous DNA subsequences of length k that are smaller than the read length. The goal of this k-mer analysis is to chop the reads into fixed-length k-mers, count the number of instances of each, and filter out those that occur with low frequency. Note that filtering out a k-mer does not eliminate the entire read, but only the portion that contains errors. The underlying assumption is that the genome has been read multiple times (30 would be common for a human dataset) so k-mers that appear in the input set with low frequency are likely to be errors. The k-mers are overlapping, one for each position in the reads, and a second goal of k-mer analysis is to filter out k-mers with more than one likely extensions on the left or right direction. This will simplify the de Bruijn graph structure in the next step of the assembly pipeline, where the de Bruijn graph is used to represent the connectivity of the k-mers. Therefore, the de Bruijn graph can be assumed to have only unique extensions for each vertex (k-mer) and thus linear connected components; branches and repeated sequences in the genome will be addressed by later stages of the pipeline. In this section we describe our basic parallel algorithm that counts k-mers and characterizes their extensions.

Counting the frequencies of each distinct k-mer involves reading file(s) that includes DNA short reads, parsing the reads into k-mers, and keeping a count of each distinct k-mer that

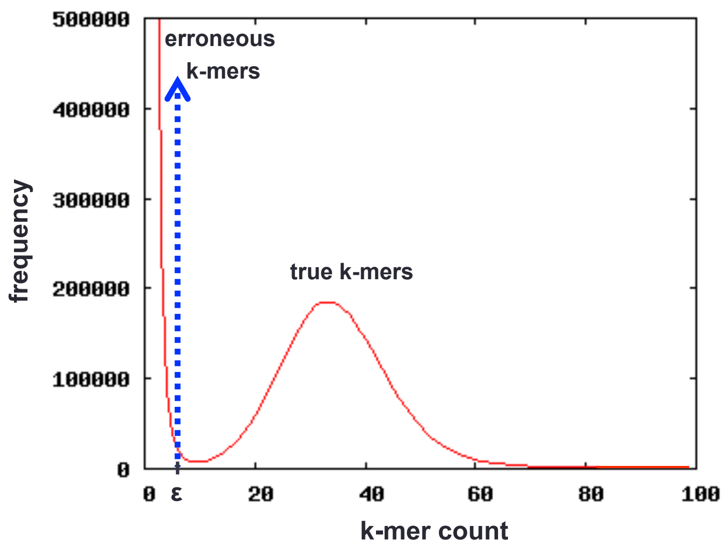


Figure 3.1: Typical k-mer histogram from a read data set.

occurs more than ϵ times ($\epsilon \approx 1, 2$). The reason for such a cutoff is to eliminate sequencing errors. Figure 3.1 shows a typical k-mer histogram from a read dataset. Observe that there are many k-mers with very small number of counts ("left" from the blue vertical arrow). The reason is that a single error in a read will create $\approx k$ erroneous k-mers: one erroneous k-mer is extracted for each one of the k initial positions of the k-length sliding window that includes that particular erroneous base in the read. Since it is quite unlikely that a similar read has exactly the same error in the exact same location, these erroneous k-mers will have a count less that ϵ.

K-mer characterization additionally requires keeping track of all possible extensions of the k-mer from either side. This is performed by keeping two short integer arrays of length four per k-mer, where each entry in the array keeps track of the number of occurrences of each nucleotide [ACGT] on either end. If a single nucleotide on an end appears more than t_{hq} times, then that end is characterized as *unique* high quality extension; the symbol U will be used throughout this dissertation to characterize such high quality extensions. On the other hand, if two (or more) nucleotides on an end appear more than t_{hq} times, then that end is designated a "fork" extension (represented with the symbol F). Finally, if no nucleotide on an end appears more than t_{hq} times, then that end is designated an "unknown" extension (represented with the symbol X). So, depending on the counts of the extensions (forward & backward) of a k-mer and a given integer constant t_{hq}, a k-mer can be characterized as UU, UF, UX, FU, FF, FX, XU, XF or XX.

One of the difficulties with performing k-mer analysis in distributed memory is that the size of the intermediate data (the set of k-mers) is significantly larger than the input, since each read is subsequenced with overlaps of $k-1$ base pairs. As each processor reads a portion of the reads, a deterministic map function maps each k-mer to a processor. This map

Algorithm 1 Parallel three-pass k-mer analysis

```
1: for all processors $p_i$ in parallel do
2:     while there are $reads$ to process do                              ▷ Pass 1
3:         $kmers$ ← ParseToKmers($reads$)
4:         $est$ ← HyperLogLog($kmers$)
5:     $globalest$ ← ParallelReduce($est$)
6:     $bfilter$ ← BloomFilter($globalest/p$)
7:     while there are $reads$ to process do                              ▷ Pass 2
8:         for a batch of $reads$ do                                      ▷ memory constraint
9:             $kmers$ ← ParseToKmers($reads$)
10:            $owners$ ← MapToProcessors($kmers$)
11:        Alltoallv($kmers, owners$)
12:        for each incoming k-mer $km$ do
13:            if $km$ exists in bfilter then
14:                Insert($locset$, $km$, $null$)
15:            else
16:                Insert($bfilter$, $km$)
17:    while there are $reads$ to process do                              ▷ Pass 3
18:        for a batch of $reads$ do                                      ▷ memory constraint
19:            $kmers$ ← ParseToKmers($reads$)
20:            $owners$ ← MapToProcessors($kmers$)
21:            $exts$ ← FetchExtensions($kmers$, $reads$)
22:        Alltoallv($kmers, owners$)
23:        Alltoallv($exts, owners$)
24:        for each incoming k-mer $km$ and extension $ext$ do
25:            if $km$ exists in locset then
26:                Update($locset$, $km$, $ext$)
```

k-$mer \rightarrow \{1, \ldots, p\}$ assigns all the occurrences of a particular k-mer sequence to the same processor, thus eliminating the need for a global hash table. Note that this computational pattern fits the Use Case 4 (LRW) described in subsection 2.3.2. Because a k-mer can be seen in two different orientations (due to reverse complementarity), the map flips it to the lexicographically smaller orientation before calculating the process number to which it is mapped.

Algorithm 1 lists our parallel k-mer analysis algorithm, which does three streaming passes, as described in detail below. During the I/O steps (lines 2, 7, and 17), each processor reads an equal amount of sequence data. The algorithm achieves almost perfect load balance in terms of the number of distinct k-mers assigned to each processor, thanks to our use of the strong MurmurHash [4] function to implement the k-mer to processor map. The *locset* is a set data structure that does not allow duplicates, hence potential reinsertion attempts (line 14) are treated as no-ops.

Unlike the later phases of our assembly pipeline, k-mer analysis is a bulk-synchronous algorithm and is written in MPI. It uses MPI's irregular personalized communication primitive, Alltoallv. Since the k-mer to processor map is uniform, as described before, the input to the Alltoallv is evenly distributed in each processor. If h is the maximum data received by any processor and there are p processors, then Alltoallv can be implemented with a communication volume of $O(h + p^2)$ per processor [5]. Unfortunately, the data received per processor, h, is not necessarily evenly distributed due to potential presence of high count k-mers. It is most problematic when there are less than p high count k-mers as there is no way to distribute them evenly in that case. In Section 3.2 we describe how our parallel algorithm deals with such high-frequency k-mers. A less synchronous algorithm might also help with modest levels of load imbalance and is a topic for future work.

3.1.1 Eliminating Erroneous k-mers

The previously mentioned deterministic k-mer to processor mapping allows us to use hash tables that are local to each processor. Even then, memory consumption quickly becomes a problem due to errors; a single nucleotide error creates $\approx k$ erroneous k-mers. It is not uncommon to have over 80% of all distinct k-mers erroneous, depending on the read length, the error-rate and the value of k. We ameliorate this problem using Bloom filters, which were previously used in serial k-mer counters [78].

A Bloom filter [8] is a space-efficient probabilistic data structure used for membership queries. It might have false positives, but no false negatives. If a k-mer was not seen before, the filter can accidentally report it as "seen". However, if a k-mer was previously inserted, the Bloom filter will certainly report it as "seen". This is suitable for k-mer counting as no real k-mers will be missed. If the Bloom filter reports that a k-mer was seen before, then we insert that k-mer to the final *locset* that does the actual counting. Our novelty is the discovery that localization of k-mers via the deterministic k-mer to processor mapping is *necessary and sufficient* to extend the benefits of Bloom filters to distributed memory.

The false positive rate of a Bloom filter is $Pr(e) = (1 - e^{-hn/m})^h$ for m being the number of distinct elements in the dataset, n the size of the Bloom filter, and h the number of hash functions used. There is an optimal number of hash functions given n and m, which is $h = \ln 2 \cdot (m/n)$. In practice, we achieve approximately 5% false positive rate using only 1-2% of the memory that would be needed to store the data directly in a hash table (without the Bloom filter). Hence, in a typical dataset where 80% of all k-mers are errors, we are able to filter out 76% of all the k-mers using almost no additional memory. Hence, we can effectively run a given problem size on a quarter of the nodes that would otherwise be required.

3.1.2 Estimating the Bloom Filter Size

We have so far ignored that Bloom filters need to know the number of distinct elements expected to perform optimally. While dynamically resizing a Bloom filter is possible, it is expensive to do so. We therefore use a cardinality estimation algorithm to approximate the number of distinct k-mers. Specifically, we use the Hyperloglog algorithm [31], which

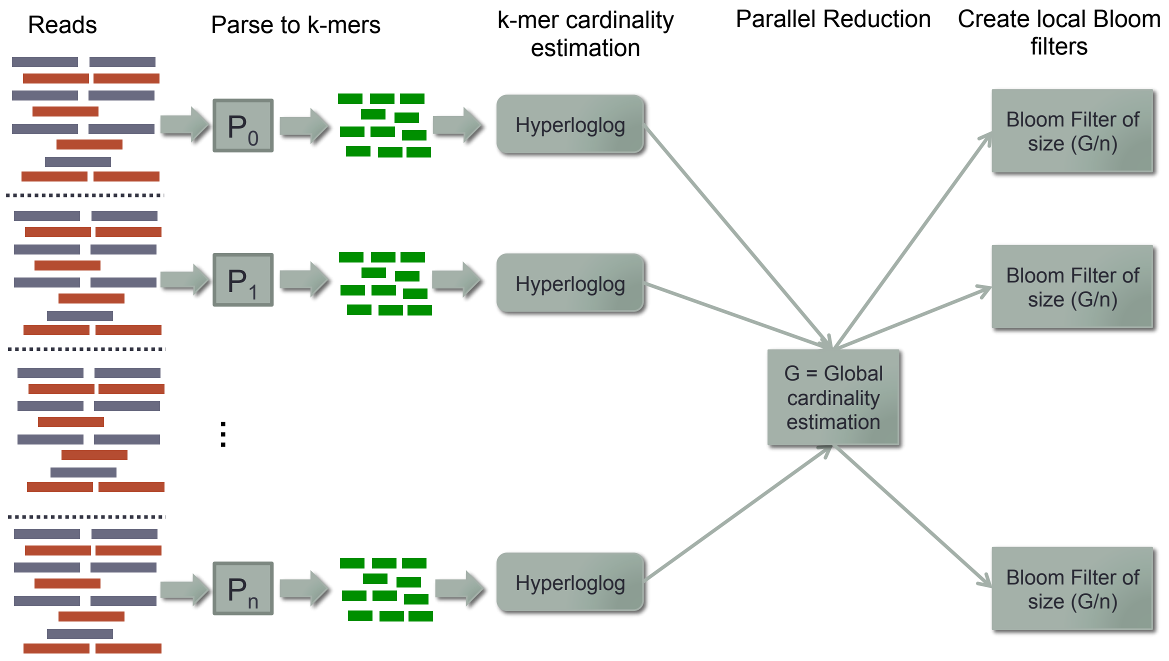

Reads
Parse to k-mers
k-mer cardinality estimation
Parallel Reduction
Create local Bloom filters
P0
P1
Pn
Hyperloglog
Hyperloglog
Hyperloglog
G = Global cardinality estimation
Bloom Filter of size (G/n)
Bloom Filter of size (G/n)
Bloom Filter of size (G/n)

Figure 3.2: First pass of the algorithm that uses the Hyperloglog algorithm to estimate the Bloom filter size.

achieves less than $1.04/\sqrt{m}$ error for a dataset of m distinct elements. Hyperloglog requires only several KBs of memory to count trillions of items. The key idea of the Hyperloglog algorithm is the observation that the cardinality of a multiset of uniformly distributed random numbers can be estimated by calculating the maximum number of leading zeros in the binary representation of each number in the multiset. If the maximum number of leading zeros observed is Z, an estimate for the number of distinct elements in the multiset is 2^Z. In the Hyperloglog algorithm, a hash function is applied to each element in the original multiset in order to obtain a multiset of uniformly distributed random numbers with the same cardinality as the original multiset. The cardinality of this randomly distributed set can then be estimated using the algorithm above.

The observation that leads to minimal communication parallelization of Hyperloglog is as follows. Merging Hyperloglog counts for multiple datasets can be done by keeping the maximum of their final buckets by a parallel reduction. Consequently, the communication volume for this first cardinality estimation pass is *independent of the size of the sequence data*, and is only a function of the Hyperloglog data structure size. In practice, we implement a modified version of the algorithm that uses 64-bit hash values as the original 32-bit hash described in the original study [31] is not able to process our massive datasets. Figure 3.2 illustrates the first pass of the algorithm that uses the Hyperloglog algorithm to estimate the Bloom filter size. After the Bloom filters have been initialized, we perform a second pass over the input dataset and store in the actual local sets only the k-mers that are reported

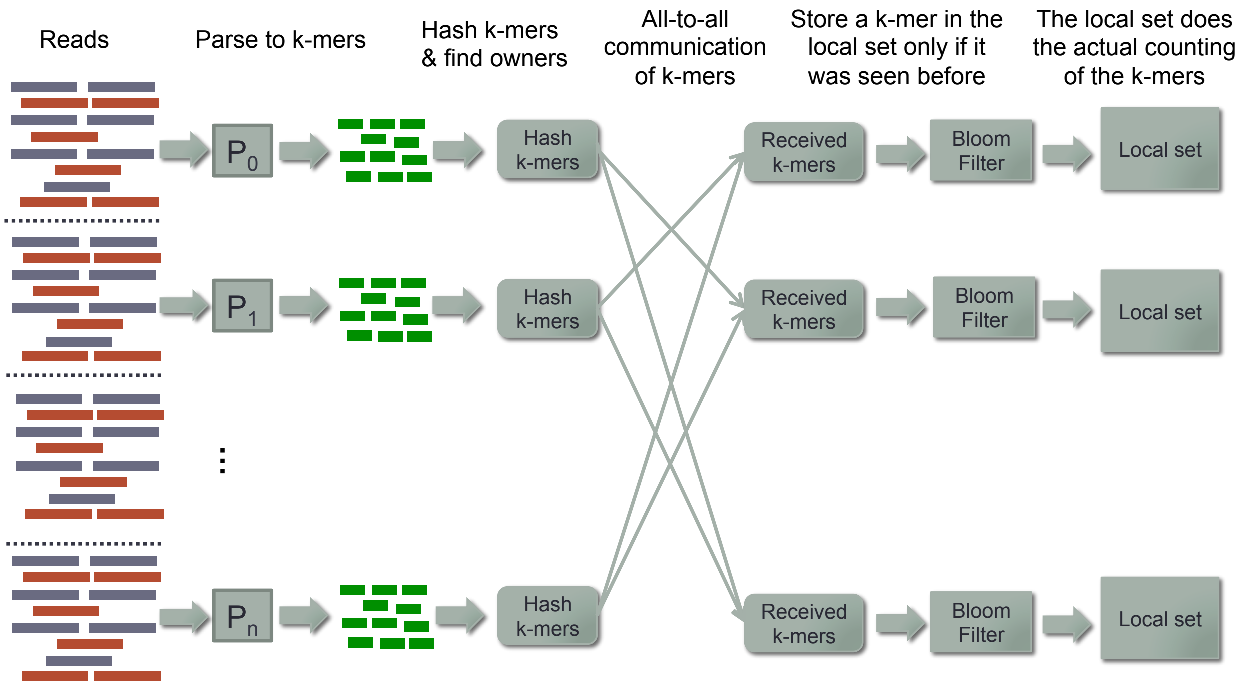

Reads
Parse to k-mers
Hash k-mers & find owners
All-to-all communication of k-mers
Store a k-mer in the local set only if it was seen before
The local set does the actual counting of the k-mers
P0
P1
Pn
Hash k-mers
Received k-mers
Bloom Filter
Local set

Figure 3.3: Second pass over the input reads where only the k-mers that are reported by the corresponding Bloom filter as “seen” are eventually stored in the local sets.

by the corresponding Bloom filter as “seen” (see Figure 3.3). Finally, with a third pass we keep track of the number of occurrences of each extension for the k-mers existing in the actual set (see Figure 3.4). The outcome of the k-mer analysis step is set of k-mers and the corresponding counts of their extensions. Given a threshold t_{hq} we can designate each extension as `U`, `F` or `X` in subsequent steps of the assembly pipeline.

3.2 High-Frequency k-mer Analysis

During the k-mer analysis phase, we assigned each k-mer to a particular “owner” processor, which counts all the occurrences of a given k-mer. In other words, processors did not perform any partial counting. This choice was largely motivated by the need to use Bloom filters for eliminating erroneous k-mers, which requires that all counting for a k-mer is performed by its owner in order to safely call it erroneous.

For a simple analysis, assume that there are n k-mers and a fraction $0 \leq f \leq 1$ of them occur only once (hence guaranteed to be an error or unreliable at best). Blindly performing partial counting in each processor for all k-mers encountered would require $O(np)$ aggregate space over p processors. Using the owner-computes model and Bloom filters reduces this overhead to $O(fn)$. The downside of this approach is that highly complex plant genomes, such as wheat, are extremely repetitive and it is not uncommon to see k-mers that occur

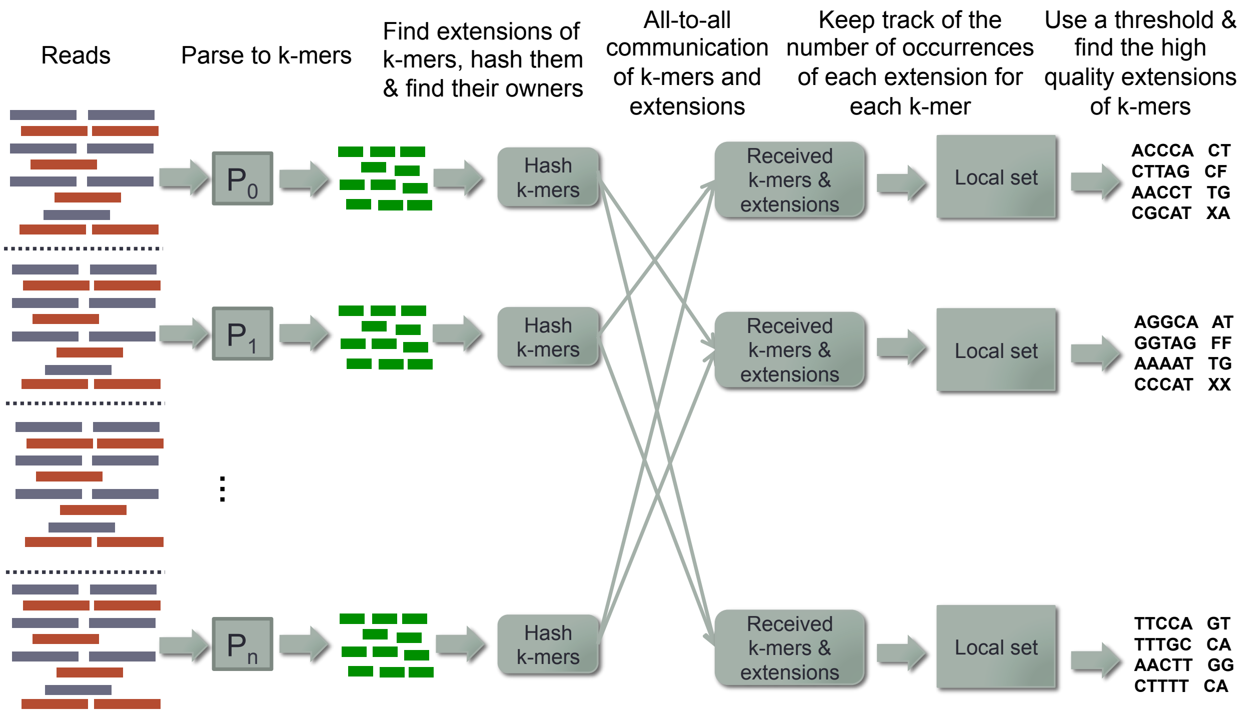

Reads
Parse to k-mers
Find extensions of k-mers, hash them & find their owners
All-to-all communication of k-mers and extensions
Keep track of the number of occurrences of each extension for each k-mer
Use a threshold & find the high quality extensions of k-mers
P0
P1
Pn
Hash k-mers
Received k-mers & extensions
Local set
ACCCA CT
CTTAG CF
AACCT TG
CGCAT XA
AGGCA AT
GGTAG FF
AAAAT TG
CCCAT XX
TTCCA GT
TTTGC CA
AACTT GG
CTTTT CA

Figure 3.4: Third pass over the input reads where we keep track of the number of occurrences of each extension for the k-mers existing in the actual set.

millions of times. For example, the hexaploid bread wheat lines "Synthetic W7984" assembled with an early version of the HipMer contig generation [20] has about 2,000 k-mers that each occur more than half a million times and about 70 k-mers that occur over 10 million times for $k = 51$. Such high-frequency k-mers create a significant load imbalance problem, as the processors assigned to these high-frequency k-mers require significantly more memory and processing times and also create imbalanced communication times.

Consequently, we improve our approach by first identifying frequent k-mers (i.e. "heavy hitters" in database literature) and treating them specially. In particular, the owner-computes method is still used for low-to-medium frequency k-mers but the high frequency k-mers are accumulated locally, followed by a final global reduction. Since an initial pass over the data is already performed to estimate the cardinality (the number of distinct k-mers) and efficiently initialize our Bloom filters, running a streaming algorithm for identifying frequent k-mers during the same pass is essentially free in terms of I/O costs.

We use the counter-based algorithm of Misra and Gries [82] (subsequently reinvented several times [23, 55]). This algorithm maintains an associative array of $\theta - 1$ entries where in each entry we store a key (the item that is counted) and its value (its count). Whenever we see an item e_i in the input stream of items (k-mers in our case), we look it up in the associative array and if e_i is already there, we increment its count. If e_i is not found and there is an empty entry in the associative array, we store it there and make its count 1. If no space is left in the associative array and the item is not there, we do not store the item e_i, but we decrement the counters of all the stored items by 1. If a counter becomes 0, the corresponding item is dropped from the associative array and the cell becomes empty.

If the true frequency of a k-mer x is $f(x)$, the Misra-Gries algorithm outputs all k-mers with $f(x) \geq 1/\theta$ using $O(\theta)$ space. Furthermore, the reported count $\widetilde{f}(x)$ is a lower bound on the actual count, i.e. $\widetilde{f}(x) \leq f(x)$. Since the items with $\widetilde{f}(x) > 1$ can not be eliminated by the Bloom filter, integration of the Misra-Gries algorithm to the k-mer analysis step does not decrease the efficiency of Bloom filters as long as we only treat k-mers with $\widetilde{f}(x) > 1$ specially. For parallelization, we take advantage of the fact that the Misra-Gries algorithm creates mergeable summaries [1] and use a high-performance implementation of the algorithm described by Cafaro and Tempesta [16].

3.3 Parallel File I/O

A standard format to represent DNA short reads is the FASTQ format, a text file that includes one read per line with another line of the same length that encodes the quality of each base pair. Unfortunately, there is no scalable way to read a FASTQ file in parallel due to its text-based nature. One commonly used approach is to create many subfiles to be read by different processors. Unfortunately this approach creates problems for data management and cannot flexibly be processed by varying numbers of processors. We overcome this barrier by implementing a parallel block FASTQ reader.

The reader first samples the FASTQ file in parallel (each processor samples about a million reads concurrently) to estimate the lengths of the read names, which can be variable across reads. The average read-name length is then used as an estimate to find splitting points across processors. Since a splitting point can be in the middle of a read, a processor p_i fast forwards to the beginning of the next full read, while the previous partial read is processed by the neighboring processing p_{i-1}. The key to high performance is to use MPI-IO functions (in our case `MPI_File_read_at`) with large buffer sizes and parse the buffered data in memory. Using this approach, our method obtains performance close to the theoretical I/O bandwidth.

3.4 Experimental Results

3.4.1 Experimental Testbed

High-concurrency experiments are conducted on Edison, a Cray XC30 supercomputer at NERSC. Edison has a peak performance of 2.57 petaflops/sec, with 5,576 compute nodes, each equipped with 64 GB RAM and two 12-core 2.4 GHz Intel Ivy Bridge processors for a total of 133,824 compute cores. The compute nodes are interconnected with the Cray Aries network using a Dragonfly topology.

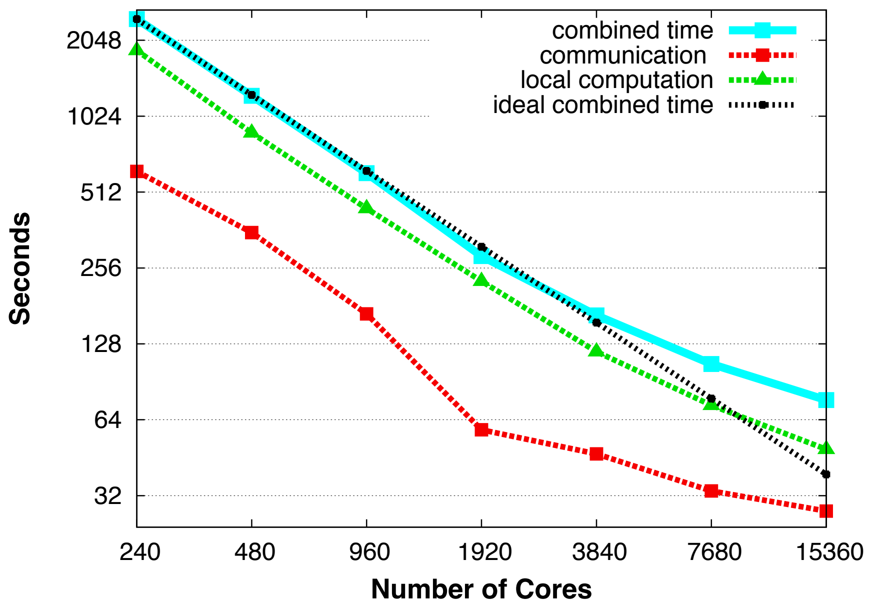


Figure 3.5: Strong scaling of the k-mer analysis on Edison for the human genome **before** the "high-frequency k-mer optimization". Both axes are in log scale. Combined time corresponds to the sum of communication and computation times of the k-mer analysis.

Our experiments are conducted on real datasets for the human and wheat genomes. The 3.2 Giga base pair (gigabase, Gb) human genome is assembled from 2.5 billion reads (252 Gbp of sequence) for a member of the CEU HapMap population (identifier NA12878) sequenced by the Broad Institute. The reads are 101 bp in length from a paired-end insert library with mean insert size 238 bp. The 17 Gbp hexaploid wheat genome (*Triticum aestivum L.*), is assembled from 2.3 billion reads (477 Gbp of sequence) for the homozygous bread wheat line "Synthetic W7984" sequenced by the JGI. The reads are 100-250 bp in length from 5 paired-end libraries with insert size 240-740 bp. This important genome was only recently sequenced for the first time [20], due to its size and complexity and our work in k-mer analysis and contig generation enabled its assembly.

3.4.2 Strong scaling results

The k-mer analysis step of the original Meraculous pipeline had 3,459 GB overall memory footprint. By contrast, our optimized k-mer analysis implementation only required 499 GB of RAM for the 240 cores run on human genome, a reduction of 6.93×. Our code achieves a rate of over 3 billion k-mers analyzed per second for our largest run (15K cores) on the human data, including the I/O cost. Our maximum attained aggregate I/O performance is 18.5 GB/s out of the theoretical peak of 48 GB/s of Edison. The ability to perform this step so fast allowed us to do previously infeasible exploratory analyses to optimize the assembly as a function of k.

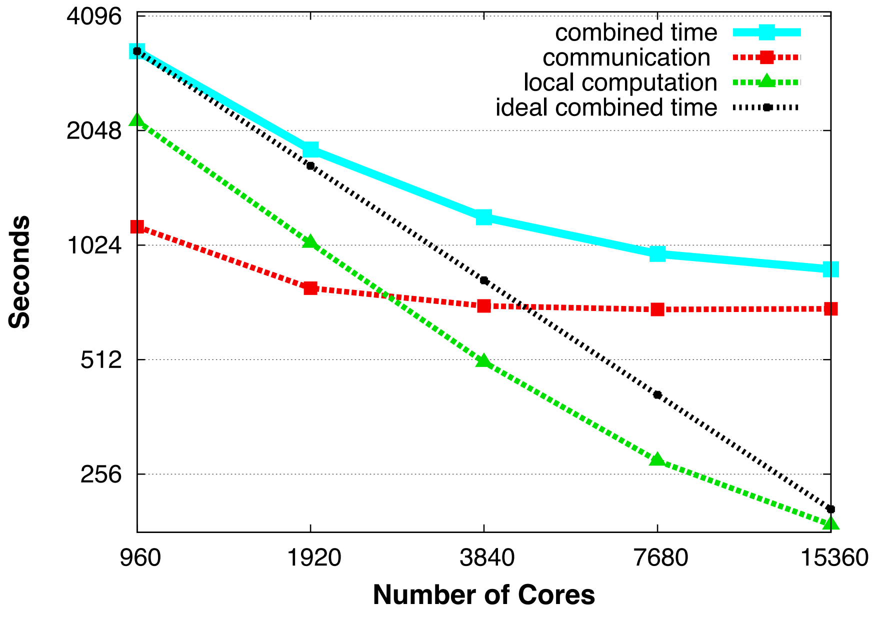


Figure 3.6: Strong scaling of the k-mer analysis on Edison for the wheat genome **before** the "high-frequency k-mer optimization".

The strong scaling results for the human and wheat data **before** the "high-frequency k-mer optimization" are shown in Figures 3.5 and 3.6 respectively. Our algorithm scales efficiently for the human data, all the way from 240 cores to over 15K cores, for both communication and local computation phases. The relatively poor scaling of communication for the wheat is because wheat data is significantly more skewed with about 60 k-mers occurring over 10 million times, and only 403 k-mers occurring over 1 million times each. This creates load imbalance in the Alltoallv phase as the receiving processors of those extreme frequency k-mers spend relatively longer time in the collective call.

Figure 3.7 presents the effect of identifying heavy hitters (i.e. "high-frequency k-mers") and avoiding communication for each of those on the wheat genome dataset. As described in Section 3.2, we treat heavy hitters specially by first accumulating their counts and extensions locally, followed by a final global reduction. We do not show the results for human as the heavy hitters optimization does not significantly impact its running time. We use $\theta = 32,000$ in our experiments, which is the number of slots in the main data structure of the Misra-Gries algorithm. Since the wheat data has approximately 330 billion 51-mers, this choice of θ only guarantees the identification of k-mers with counts above 10 million. In practice, however, the performance was not sensitive to the choice of θ, which was varied between 1K and 64K with negligible (less than 10%) performance difference. At the scale of 15,360 cores, the heavy hitters optimization results in a 2.4× improvement for the wheat data.

In the default version, the percentage of communication increases from 23% in 960 cores to 68% in 15,360 cores and as a consequence we observe poor strong-scaling. In the optimized

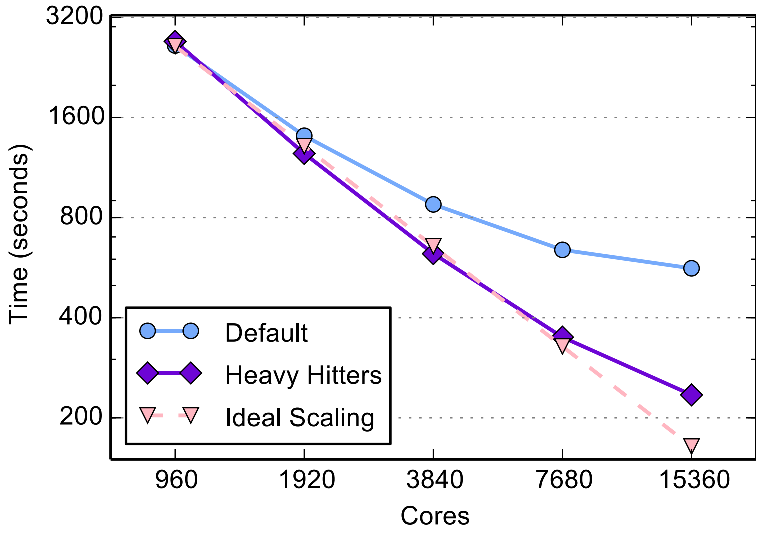


Figure 3.7: Strong scaling of k-mer analysis on the wheat genome **after** the heavy hitters ("high-frequency k-mers") optimization.

version, however, communication increases from 16% to only 22%, meaning that the method is no longer communication bound for the challenging wheat dataset and is showing close to ideal scaling up to 7,680 processors. Scaling beyond that is challenging because the run times include the overhead of reading the FASTQ files, which requires 40-60 seconds independently of the number of processors (depending on the system load). Since the Edison I/O bandwidth is already saturated by 960 cores, the I/O costs are relatively flat with increasing number of cores and thus impact scalability at the highest concurrency.

3.5 Conclusion

In this Chapter we presented a distributed memory algorithm for k-mer analysis. Our algorithm uses Bloom filters to reduce the memory footprint by 7×. Also, in order to mitigate the load imbalance caused by heavy hitters we incorporated a streaming algorithm that identifies the high-frequency k-mers and deals with them separately; this optimization yields a speedup up to 2.4× in a dataset with highly skewed distribution of k-mers. Experimental results on the Edison supercomputer illustrate efficient scaling up to 15K cores, enabling us to do previously infeasible exploratory analyses for different values of k.

Chapter 4

Parallel Algorithms for Contig Generation

In this Chapter[1] we present parallel algorithms for the second stage of the assembly pipeline, namely contig generation. First, we present the parallel algorithm and optimizations for the de Bruijn graph construction, a special graph that is used to represent overlap among k-mers. Then, we present the parallel algorithm that traverses the de Bruijn graph and identifies connected components that constitute the contigs. Our algorithm is work-optimal, exhibits massive parallelism and employs a lightweight synchronization scheme. We also introduce a communication-avoiding optimization which further improves the performance under various use cases. Finally, we present large-scale experimental results on Edison and conclude the parallel contig generation algorithms.

4.1 Parallel de Bruijn Graph Construction

We now discuss our parallel de Bruijn graph construction algorithm, where the de Bruijn graph represents the connectivity of the k-mers. In particular we are working with a linear subgraph of the full de Bruijn graph, as specified by the Meraculous approach in Section 2.1 because we are using only the `UU` k-mers from the previous k-mer analysis step and therefore we have excluded all the "fork" structures in the graph. Below, we use the term "de Bruijn graph" to refer to this linear subgraph of the full de Bruijn graph where only the `UU` k-mers are included. The branches of the full de Bruijn graph will be addressed by later stages of the pipeline.

In our approach a de Bruijn graph is represented as a hash table with a k-mer as "key" (a k-mer is a vertex in the graph) and a two-letter code `[ACGT][ACGT]` as "value" which represents two undirected edges incident to that vertex. The parallelization relies heavily on a high

[1]The material presented in this Chapter was first published in the papers "Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly" [35] and "HipMer: An Extreme-Scale De Novo Genome Assembler" [33].

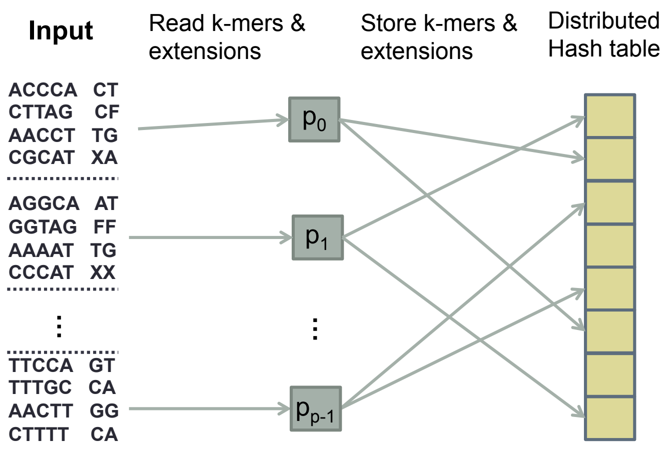


Figure 4.1: Parallel de Bruijn graph construction. Only `UU` k-mers will be stored in the distributed hash table and input k-mers with an `F` or `X` extension are ignored during this stage.

performance distributed hash table, shown in Figure 4.1 that inputs a set of k-mers with their corresponding two letter code (high quality extensions). Assuming m initial k-mers and p processors, each processor will read m/p k-mers, hash the keys and store only the input `UU` entries to the appropriate buckets of the distributed hash table.

4.1.1 Dynamic message aggregation

The basic algorithm described in the previous subsection suffers from fine-grained communication and synchronization required to store k-mers into the distributed hash table. Multiple processors might try to store simultaneously k-mers in the same bucket and therefore a synchronization mechanism is required to ensure atomicity and prevent race conditions. Note that the computational task of the graph construction is to insert all the `UU` k-mers in a distributed hash table that can be globally accessed for later use. We recognize this computational pattern as the Use Case 1 (GUO) of the distributed hash tables (see subsection 2.3.2), therefore we can mitigate the communication and synchronization overheads by leveraging dynamic message aggregation called *aggregating stores*, shown in Figure 4.2. Here, a processor p_i has $p-1$ local buffers corresponding to the rest $p-1$ remote processors, where the size S of each local buffer is a tuning parameter. Every processor hashes a k-mer entry and calculates the location in the hash table where it has to be stored. Instead of incurring a remote access to the distributed hash table, the processor computes the processor ID owning that remote bucket in the hash table and stores the entry to the appropriate *local* buffer.

In this approach, when a local buffer dedicated for processor p_j becomes full, a remote aggregate transfer to the processor p_j is initiated. Each processor p_j has a pre-allocated shared space (henceforth called *local-shared stack*) where other processors can store entries destined for that processor p_j. In Section 4.2.4 we describe the memory management of the

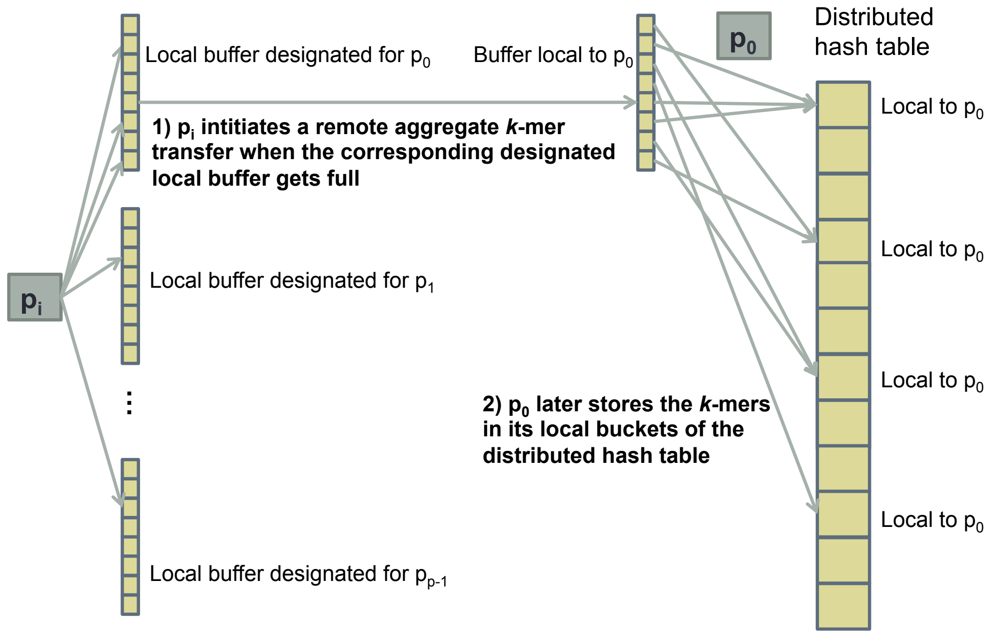

Local buffer designated for p_0
Buffer local to p_0
p_0
Distributed hash table
Local to p_0
1) p_i intitiates a remote aggregate *k*-mer transfer when the corresponding designated local buffer gets full
p_i
Local buffer designated for p_1
Local to p_0
Local to p_0
2) p_0 later stores the *k*-mers in its local buckets of the distributed hash table
Local to p_0
Local buffer designated for p_{p-1}

Figure 4.2: Dynamic message aggregation for the de Bruijn graph construction. In this example, processor p_i performs one remote aggregate transfer to processor p_0 when the local buffer for p_0 gets full. p_0 will store these k-mers in its local buckets later by iterating over its local-shared stack.

local-shared stacks. Once all k-mers are processed, each processor iterates over its local-shared stack and stores each entry to the appropriate *local* bucket in the distributed hash table, in a communication-free fashion. This optimization trades off $S \times (p-1)$ extra memory for the reduction in the number of messages by a factor of S compared to the unoptimized version, while the communication volume remains the same. This algorithm also avoids data races on the hash table by having the owner process do all the inserts into the hash table. At the completion of this process, the so called `UU` k-mers are stored in the the distributed hash table.

4.2 Parallel de Bruijn Graph Traversal

Given the de Bruijn graph construction, we now describe our parallel algorithm used to traverse the graph, find the corresponding linear connected components and output the computed set of contigs (defined in Section 2.1).

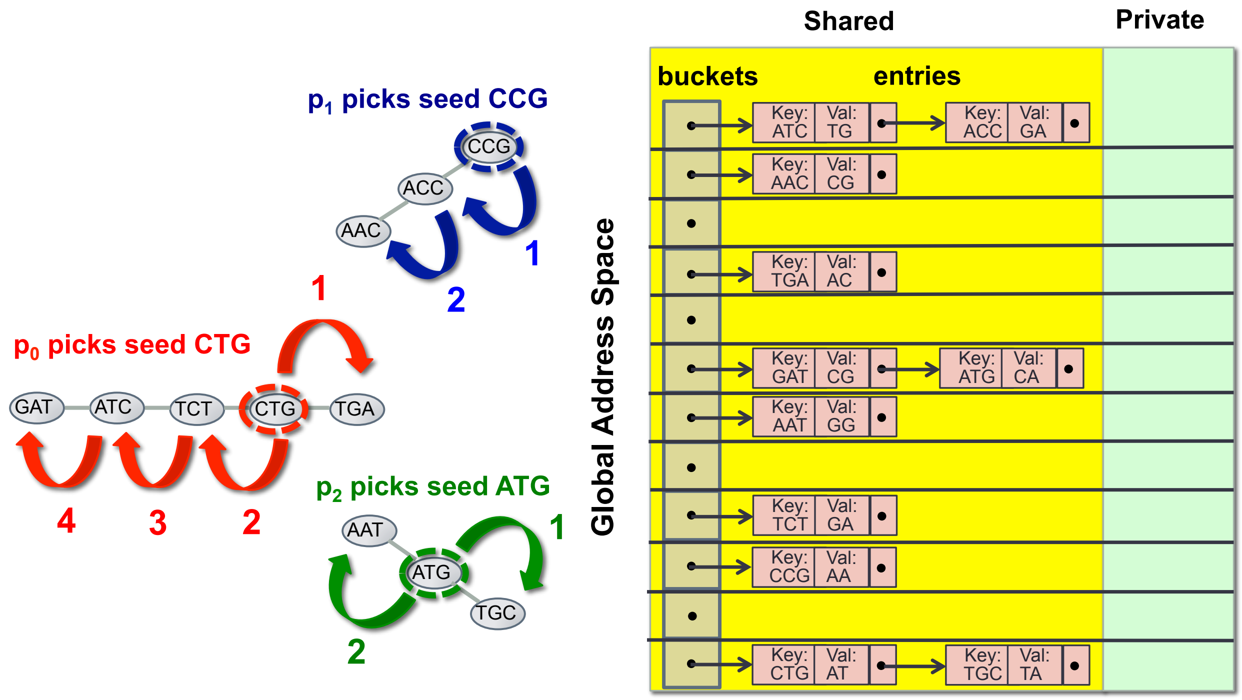


Figure 4.3: Parallel de Bruijn graph traversal. Processor 0 picks a k-mer called "traversal seed" (vertex `CTG`) and with four lookups in the distributed hash tables it explores the four remaining vertices of that connected component. The red numbered arrows indicate the order in which processor 0 looks up the corresponding vertices in the distributed hash table. In an analogous way, processors 1 and 2 pick seeds `CCG` and `ATG` respectively and explore in parallel with processor 0 different connected components of the underlying de Bruijn graph.

4.2.1 Parallel Traversal Without Conflicts

In order to form a contig, a processor p_i chooses a random k-mer from its own part of the distributed hash table as seed and creates a new *subcontig* (incomplete contig) data structure which is represented as a string and the initial content of the string is the seed k-mer. Processor p_i then attempts to extend the subcontig towards both of its endpoints using the high quality extensions stored as values in the distributed hash table. To extend a subcontig from its right endpoint, processor p_i uses the $k-1$ last bases and the right high quality extension R from the right-most k-mer in the subcontig. It therefore concatenates the last $k-1$ bases and the extension R to form the next k-mer to be searched in the hash table. Processor p_i performs a lookup for the newly formed k-mer and if it is found successfully, the subcontig is extended to the right by the base R. The same process can be repeated until the lookup in the hash table fails, meaning that there are no more `UU` k-mers that could extend this subcontig in the right direction. A subcontig can be extended to its left endpoint using an analogous procedure. If processor p_i can not add more bases to either endpoint of the subcontig, then a contig has been formed (or equivalently a connected component in the de Bruijn graph has been explored) and is stored accordingly.

Figure 4.3 illustrates how the parallel algorithm works with three processors. Processor 0 picks a random traversal seed (vertex `CTG`) and initializes a subcontig with content `CTG`. Then, by looking in the distributed hash table the entry `CTG` it gets back the value `AT`, meaning that the right extension is `A` and the left extension is `T`. After that, processor 0 forms the next k-mer to be looked up (`TGA`) by concatenating the last 2 bases of `CTG` and the right extension `A` – this lookups corresponds to the red arrow with number 1. By following the analogous procedure and three more lookups in the distributed hash table, processor 0 explores all the vertices of that connected component that corresponds to the contig `GATCTGA`. The red numbered arrows indicate the order in which processor 0 looks up the corresponding vertices in the distributed hash table. In an analogous way, processors 1 and 2 pick seeds `CCG` and `ATG` respectively and explore in parallel with processor 0 different connected components of the underlying de Bruijn graph.

4.2.2 Lightweight Synchronization Scheme

All processors independently start building subcontigs and no synchronization is required unless two processors pick initial k-mer seeds that eventually belong in the same contig. In this case, the processors have to collaborate and resolve this conflict in order to avoid redundant work. We now explain our lightweight synchronization scheme at the heart of the parallel de Bruijn graph traversal.

Basic Data Structures

Before detailing the synchronization scheme, we describe the high-level enhancements required in the data structures. Figure 4.4 illustrates these basic data structures and the way they are coupled. The k-mer data structure includes the k-mer sequence, the forward/backward extensions as well as a binary flag `used_flag` that may take two possible values: `UNUSED` or `USED`. Initially this field is set to the `UNUSED` value. When a processor finds a k-mer in the hash table:

- The processor reads the k-mer's `used_flag`.
- If the value is `UNUSED` then the processor can infer that no other processors have reached this k-mer and thus can visit it. It therefore sets that k-mer flag to `USED` and can use the k-mer's information conflict-free.
- If the value is `USED` then another processor has already added that k-mer to another subcontig.

Note that these actions need to be executed in an atomic fashion to ensure correctness (detailed in Section 4.2.4). Thus the k-mer's `used_flag` is updated via `compare_and_swap()` remote atomics to avoid races and redundant work. Furthermore, the k-mer data structure

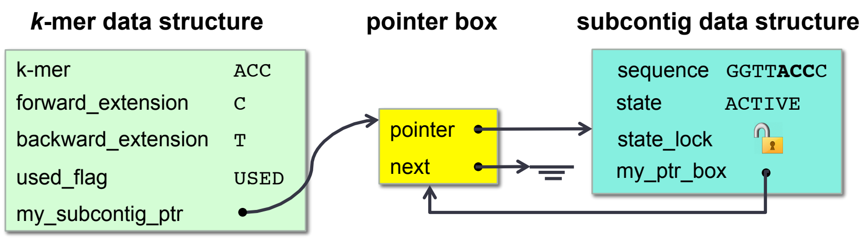


Figure 4.4: Basic data structures used in the de Bruijn graph traversal. In this example, the k-mer's `used_flag` is `USED` since the k-mer has been added to a subcontig. The `my_subcontig_ptr` field points to the `pointer box` which contains another `pointer` to the subcontig's data structure that the k-mer currently belongs to. The `next` field is a pointer to a list of box pointers; this list represents other subcontigs that have been potentially attached to the current subcontig. The `state` flag of the subcontig is `ACTIVE` since a processor is working on it. The `my_ptr_box` field has a pointer that points back to the pointer box of the current subcontig.

has a pointer `my_subcontig_ptr` to a `pointer box` containing another `pointer` to the subcontig's data structure that the k-mer currently belongs to. The `pointer box` has also a `next` pointer that points to a list of pointer boxes. In the next paragraphs we detail how these lists of pointer boxes are used in the synchronization scheme. This level of indirection is introduced for efficiency reason as it will be explained later in this section.

The subcontig data structure includes (i) the subcontig's sequence, (ii) a lock `state_lock`, (iii) a `my_ptr_box` field that points to the pointer box of the current subcontig and (iv) a flag `state` that may take one of the following values: `ACTIVE`, `ABORTED` and `COMPLETE`.

Synchronization Algorithm

The synchronization scheme in our parallel de Bruijn graph traversal is based on the subcontig's state machine illustrated in Figure 4.5. Note that a processor might conflict with at most two processors due to the one dimensional nature of the subcontigs. Therefore, when we mention neighboring subcontigs of a subcontig S we refer the subcontigs that conceptually lie to the right and to the left of S in the eventual contig. Similarly, when we refer to the neighboring processors of p_i we are referring to the processors that are working on the neighboring subcontigs.

When a processor p_i picks a k-mer as traversal seed, it also creates a new subcontig data structure and sets the state to `ACTIVE` (see arrow $\textcircled{1}$ in Figure 4.5); the subcontig also contains the selected k-mer string as the initial value of the subcontig's sequence. The seed's `my_subcontig_ptr` is set to point to a pointer box that contains another pointer to the newly created subcontig. The `my_ptr_box` field of the subcontig points back to the pointer box. The processor p_i tries to extend the subcontig in both directions using the previously described procedure.

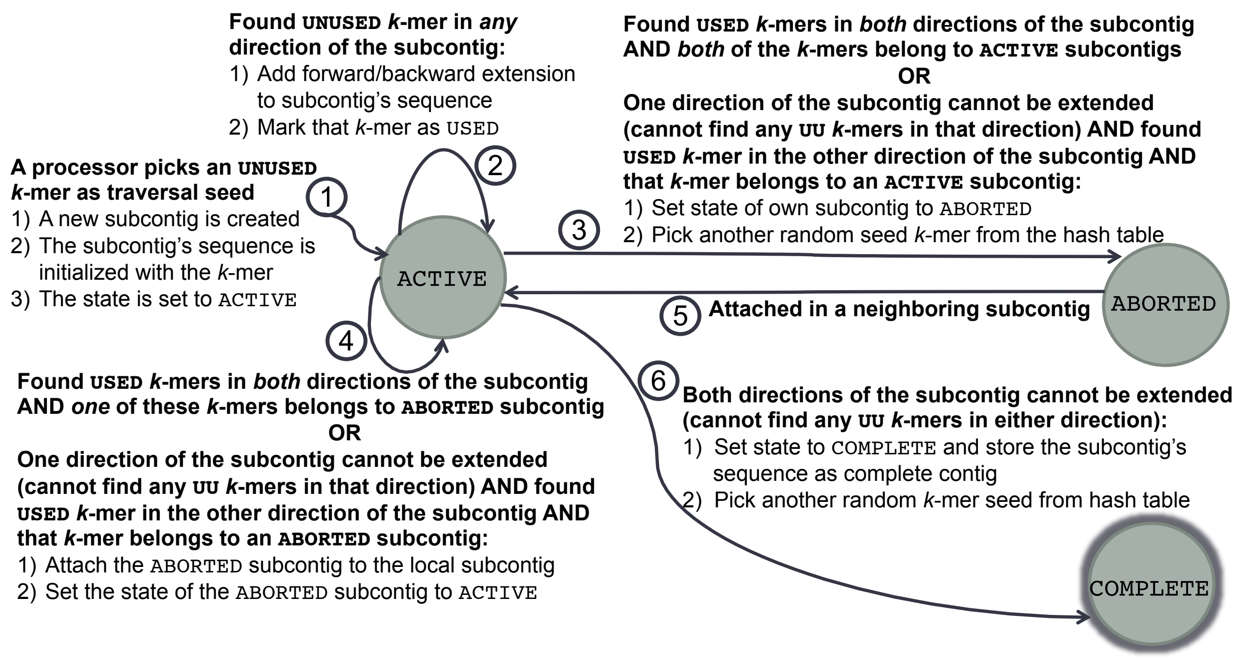

Found UNUSED k-mer in any direction of the subcontig:
1) Add forward/backward extension to subcontig's sequence
2) Mark that k-mer as USED
Found USED k-mers in both directions of the subcontig AND both of the k-mers belong to ACTIVE subcontigs
OR
One direction of the subcontig cannot be extended (cannot find any UU k-mers in that direction) AND found USED k-mer in the other direction of the subcontig AND that k-mer belongs to an ACTIVE subcontig:
1) Set state of own subcontig to ABORTED
2) Pick another random seed k-mer from the hash table
A processor picks an UNUSED k-mer as traversal seed
1) A new subcontig is created
2) The subcontig's sequence is initialized with the k-mer
3) The state is set to ACTIVE
1
2
3
4
5
6
ACTIVE
ABORTED
COMPLETE
Attached in a neighboring subcontig
Found USED k-mers in both directions of the subcontig AND one of these k-mers belongs to ABORTED subcontig
OR
One direction of the subcontig cannot be extended (cannot find any UU k-mers in that direction) AND found USED k-mer in the other direction of the subcontig AND that k-mer belongs to an ABORTED subcontig:
1) Attach the ABORTED subcontig to the local subcontig
2) Set the state of the ABORTED subcontig to ACTIVE
Both directions of the subcontig cannot be extended (cannot find any UU k-mers in either direction):
1) Set state to COMPLETE and store the subcontig's sequence as complete contig
2) Pick another random k-mer seed from hash table

Figure 4.5: Subcontig's state machine used for synchronization during the parallel de Bruijn graph traversal. Boldface text indicates the preconditions for the corresponding transition. Regular text indicates the actions that occur during the transition. We emphasize that a processor can read the neighboring subcontigs' state only **after** obtaining all the corresponding `state_lock`s in an order indicated by the processors' ids.

When p_i finds `UNUSED` k-mers in *either* direction of the subcontig, it successfully adds the forward/backward extension bases to the subcontig's sequence and marks the visited k-mers' `used_flag` field as `USED` (see arrow $\textcircled{2}$). At the same time, p_i updates the visited k-mers' `my_subcontig_ptr` field to point to the pointer box pointing to the subcontig's data structure. Meanwhile, the subcontig's `state` remains `ACTIVE` (see Figure 4.4).

Assume that processor p_i is unable to extend its current subcontig in *any* direction because it has either found `USED` k-mers in both directions or it cannot find any `UU` k-mer in one direction and has found a `USED` k-mer in the other direction. Processor p_i then attempts to obtain *all* the `state_lock`s of the neighboring subcontigs including its own `state_lock` in a *total global order* indicated by the processors' ids — and thus ensuring that deadlocks are avoided. Processor p_i has access to the neighboring subcontigs' data structures (and consequently to the appropriate `state_locks`) via the `my_subcontig_ptr` fields of the `USED` k-mers found in either direction. The processor ids can be extracted from the global pointers stored in the pointer boxes.

After obtaining these `state_locks`, p_i examines the `state` fields of the neighboring subcontigs and takes appropriate actions as indicated by the state machine in Figure 4.5:

- If both neighboring subcontigs are `ACTIVE` or one neighboring subcontig is `ACTIVE` and

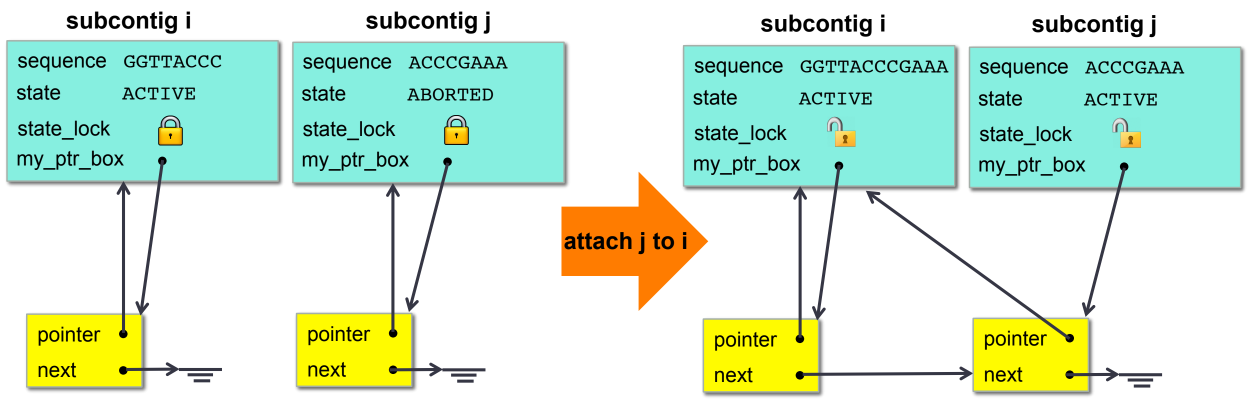


Figure 4.6: The process of attaching subcontigs. In this example, subcontig j is `ABORTED` and will be attached to subcontig i that is `ACTIVE`. First, processor p_i concatenates to the sequence of subcontig i the non-overlapping part of the sequence of subcontig j (note that the sequences of subcontigs i and j overlap by $k+1$ bases). Then, processor p_i owning subcontig i visits the pointer box list in the `my_ptr_box` field of subcontig j and updates all the `pointer` fields to point to subcontig i. Also, the pointer box list of subcontig i is concatenated with the pointer box list of subcontig j. Finally, p_i sets the `state` of the "aborted" subcontig to `ACTIVE` as it is now added to an "active" subcontig.

the other direction cannot be extended (because we cannot find any `UU` k-mers in that direction), then p_i sets the `state` of its subcontig to `ABORTED`, releases the obtained `state_lock`s following the inverse order in which they were obtained and picks another random k-mer seed from the distributed hash table to initiate another traversal (see arrow ③).

- If at least one of the neighboring subcontigs is `ABORTED` then p_i attaches that subcontig to its own (see arrow ④).

  Figure 4.6 illustrates the attaching process. In this example, subcontig j is `ABORTED` and will be attached by processor p_i to subcontig i that is `ACTIVE`. Note that the sequences of subcontigs i and j overlap by $k+1$ bases. First, processor p_i concatenates to the sequence of subcontig i the non-overlapping part of the sequence of subcontig j. Then, processor p_i visits the pointer box list in the `my_ptr_box` field of subcontig j and updates all the `pointer` fields to point to subcontig i. Also, the pointer box list of subcontig i is concatenated with the pointer box list of subcontig j. Afterwards, p_i sets the `state` of the "aborted" subcontig to `ACTIVE` as it is now added to an "active" subcontig (see arrow ⑤). Finally, p_i releases the obtained `state_lock`s following the inverse order in which they were obtained and continues to extend its subcontig.

  All the k-mers of the previously "aborted" subcontig have access (indirectly through the updated pointer boxes) to the pointer of the new subcontig where they have been attached. Such a scheme with a level of indirection allows us to avoid revisiting all these k-mers.

A processor working on an `ACTIVE` subcontig A continues computing while other processors might obtain A's `state_lock` and examine A's `state`, hence allowing our synchronization scheme to be lightweight. Additionally, a processor "aborts" its subcontig when there are more processors working on the same eventual contig and they will claim the "aborted" subcontig later, hence avoiding redundant work. Note that the subcontigs' `state` flag does not require atomic update since the locking scheme we describe guarantees that at most one processor will examine and update a subcontig's state flag at any point in time.

When both directions of the subcontig cannot be extended (cannot find any `UU` k-mer in either direction), p_i sets the state of the subcontig to `COMPLETE` and stores the subcontig's sequence as a complete contig (see arrow $\textcircled{6}$). Afterwards, p_i picks another random k-mer seed from the distributed hash table and initiates another traversal. When all k-mers in the hash table have been visited the parallel de Bruijn graph traversal is complete.

The computational task of the graph traversal is to visit all the already inserted k-mers in the distributed hash table. During this parallel process, we can't batch reads and/or writes since there might be race conditions that affect the control flow of the synchronization algorithm. However, we use global atomics instead of fine-grained locking and we build synchronization protocols at a higher level that do not involve the distributed hash table directly but instead are triggered by the results of the atomic operations on the objects stored inside the hash table. We recognize this computational pattern as the Use Case 2 (GRW) of the distributed hash tables (see subsection 2.3.2).

4.2.3 De Bruijn Graph Traversal Synchronization Cost Analysis

Since the synchronization cost of the parallel de Bruijn graph traversal will likely determine parallel efficiency, we present three analytical propositions to quantify the correlation between the expected subcontig conflicts and the concurrency. Proposition 1 gives a lower bound on the total expected conflicts while Proposition 2 provides an upper bound. Finally Proposition 3 demonstrates the expected number of conflicts incurred on the critical path.

Proposition 1. *Let p be the total processors during parallel execution and n the number of contigs to be assembled during the de Bruijn graph traversal. Assuming that the n contigs have the same length and $n \gg p$, the number $f(p)$ of expected conflicts during the traversal increases at least linearly with p, i.e. $f(p) \geq a \cdot p + b$ for some constants a and b.*

Proof. Initially there are n contigs to be assembled and the p available processors pick random k-mers as seeds to start the traversal. A conflict occurs if two (or more) processors pick an initial seed belonging to the same contig. We can formulate this process as tossing p balls into n bins where the tosses are uniformly at random and independent of each other and thus the probability that a ball falls into any given bin is $1/n$. Let's denote $\mathcal{Q}_{i,t}$ the event where ball i falls into bin t (and thus $\Pr[\mathcal{Q}_{i,t}] = 1/n$) and Φ_{ij} be the event that ball i

and ball j collide. By using the Bayes rule we can calculate the probability of any two balls i and j falling into one bin (i.e. balls i and j collide):

$$\begin{aligned}\Pr[\Phi_{ij}] &= \sum_{t=1}^{n} \Pr[\mathcal{Q}_{j,t}|\mathcal{Q}_{i,t}] \Pr[\mathcal{Q}_{i,t}] = \sum_{t=1}^{n} \frac{1}{n} \Pr[\mathcal{Q}_{i,t}] \\ &= \frac{1}{n} \sum_{t=1}^{n} Pr[\mathcal{Q}_{i,t}] = \frac{1}{n} \sum_{t=1}^{n} \frac{1}{n} = \frac{1}{n}\end{aligned} \tag{4.1}$$

where we have used the fact that $\Pr[\mathcal{Q}_{j,t}|\mathcal{Q}_{i,t}] = \Pr[\mathcal{Q}_{j,t}] = 1/n$ since the events $\mathcal{Q}_{j,t}$ and $\mathcal{Q}_{i,t}$ are independent.

Let X_{ij} be the indicator random variable of Φ_{ij}, i.e. $X_{ij} = 1$ if Φ_{ij} happens and $X_{ij} = 0$ otherwise. We can consider X_{ij}s as Bernoulli variables and tosses can be considered as a sequence of Bernoulli trials with a probability $1/n$ of success, i.e. $\Pr[X_{ij} = 1] = \Pr[\Phi_{ij}] = 1/n$. If C is defined as the number of conflicts then $C = \sum_{i<j} X_{ij}$. So, we can calculate the expected number of conflicts $\mathbb{E}[C]$ as:

$$\mathbb{E}[C] = \sum_{i<j} \mathbb{E}[X_{ij}] = \sum_{i<j} \Pr[X_{ij} = 1] = \frac{1}{n}\binom{p}{2} \tag{4.2}$$

Now, consider the traversal as a sequence of steps, where at each step p contigs are calculated, and the traversal consists of n/p such steps. In the first step, the expected number of conflicts is given by equation 4.2: $\mathbb{E}[C_1] = \frac{1}{n}\binom{p}{2}$. We assume that as soon as these conflicts are resolved no more conflicts occur at the same step and thus $\mathbb{E}[C_1]$ is a lower bound of the expected number of conflicts in the first step. In the i-th step, there are $n-(i-1)p$ remaining contigs and we can consider the i-th step as tossing p balls into $n-(i-1)p$ bins, thus we get that $\mathbb{E}[C_i] = \frac{1}{n-(i-1)p}\binom{p}{2}$ where C_i is the number of conflicts at step i. A lower bound on the expected number of conflicts for the entire graph traversal is therefore:

$$\begin{aligned}\mathbb{E}[C_{total}] &= \mathbb{E}[C_1] + \mathbb{E}[C_2] + \cdots + \mathbb{E}[C_{n/p}] \\ &= \frac{1}{n}\binom{p}{2} + \frac{1}{n-p}\binom{p}{2} + \cdots + \frac{1}{p}\binom{p}{2} \\ &= \binom{p}{2} \cdot \frac{1}{p} \cdot \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n/p}\right) \\ &= \binom{p}{2} \cdot \frac{1}{p} \cdot H_{n/p} = \frac{p-1}{2} \cdot H_{n/p}\end{aligned} \tag{4.3}$$

where with H_i we denote the i-th partial sum of the diverging harmonic series. For large values of n/p we can write:

$$H_{n/p} \simeq \ln(n/p) + \gamma = \ln(n) - \ln(p) + \gamma \tag{4.4}$$

where $\gamma \simeq 0.577$ is the Euler-Mascheroni constant. So, the number $f(p)$ of the expected conflicts as a function of p is:

$$f(p) = \mathbb{E}[C_{total}] = \frac{p-1}{2} \cdot (\ln(n) - \ln(p) + \gamma) \tag{4.5}$$

We have assumed that $n \gg p$. If $n \geq 100 \cdot p$ then:

$$\begin{aligned} \ln(n) \geq \ln(100 \cdot p) = \ln(100) + \ln(p) \geq 4.6 + \ln(p) \\ \Leftrightarrow \ln(n) - \ln(p) + \gamma \geq 5.177 \end{aligned} \tag{4.6}$$

By combining equation 4.5 and inequality 4.6 we get that:

$$\begin{aligned} f(p) = \frac{p-1}{2} \cdot (\ln(n) - \ln(p) + \gamma) \geq \frac{p-1}{2} \cdot 5.177 \\ \Rightarrow f(p) \geq 2 \cdot p - 3 \end{aligned} \tag{4.7}$$

which completes the proof by setting $a = 2$ and $b = -3$. □

Proposition 2. *Let G be a de Bruijn graph and assume that the result of its traversal consists of a single contig L. Let C_G be the number of conflicts in this traversal by using a seeding function F_{seed}. Now assume that some nodes are removed from G resulting in a graph G'. The traversal of G' that uses the same seeding function F_{seed} results in $C_{G'}$ conflicts where $C_{G'} \leq C_G$.*

Proof. By removing a number of nodes from G, the traversal of the new graph G' results in a set of contigs S that span part of the single contig L. More accurately, the only places that the combination of S's members differs from L are those ones that correspond to the removed nodes (k-mers). Now consider a seeding function F_{seed} that incurs C_G conflicts in G. The same seeding function F_{seed} creates at most C_G conflicts in G' because the removed nodes only can prevent a conflict (since a removed node divides a contig in two independent subcontigs). Therefore we get: $C_{G'} \leq C_G$. □

The result of Proposition 2 suggests that the worst case for the number of conflicts in a traversal is one in which the resulting assembly is a single contig and thus C_G is an upper bound in the number of conflicts. Now, let's assume that the seeds are picked uniformly distanced across the single contig L and that p processors are performing the traversal of G in parallel. Such a traversal of G results in $O(p)$ conflicts. By using the latter fact and the result of Proposition 2 we conclude that an upper bound for the total number of conflicts during a traversal with seeding function resulting in uniformly distanced seeds is $O(p)$. Also, Proposition 1 indicates that the total number of conflicts is $\Omega(p)$. Therefore, under the assumptions made in Propositions 1 and 2, the total number of expected conflicts is $\Theta(p)$.

Let T_{serial} be the serial time for de Bruijn graph traversal, p the number of total processors, $\mathcal{C}_{crit}(p)$ the expected number of conflicts incurred on the critical path and t_{confl} the effective time for resolving a conflict. Then the time for the parallel traversal $T(p)$ can be modeled roughly as:

$$T(p) = \frac{T_{serial}}{p} + \mathcal{C}_{crit}(p) \cdot t_{confl} \tag{4.8}$$

The first term of Equation 4.8 decreases linearly with p and is the useful work, while the second term is the synchronization overhead of the parallel algorithm, and $\mathcal{C}_{crit}(p)$ depends on the total number of conflicts $\mathcal{C}_{total}(p)$ and p. We showed that under some assumptions $\mathcal{C}_{total}(p) = \Theta(p)$ and thus a synchronization scheme that merely leaves $\Theta(p)$ conflicts to be resolved on the critical path is unacceptable because then the second term of equation 4.8 would increase linearly with p. The synchronization algorithm in subsection 4.2.2 is lightweight in a sense that as soon as a conflict is detected, one of the involved processors takes actions according to the state machine and does not prevent others from doing useful work. Proposition 3 shows that the expected number of the conflicts incurred on the critical path is $\mathcal{C}_{crit}(p) = 2 \cdot \mathcal{C}_{total}(p)/p$.

Proposition 3. *Let $\mathcal{T}$ be the set of conflicts during the parallel de Bruijn graph traversal with cardinality $|\mathcal{T}| = \mathcal{C}_{total}(p)$. Assuming that the eventual contigs have the same length, the expected number of the conflicts incurred along the critical path $\mathcal{C}_{crit}(p)$ is equal to $2 \cdot \mathcal{C}_{total}(p)/p$.*

Proof. We will use the same formulation of the problem as in Proposition 1. Additionally consider a collision in a particular bin t during a step of the algorithm where there are R bins (equivalently consider a conflict in a particular contig t during a step of the traversal where there are R remaining contigs to be assembled). Let $\mathcal{Q}_{i,t}$ be the same as in Proposition 1, Z_t the indicator variable of the event "there is a collision in bin t" and Φ^t_{ij} the event that balls i and j fall in bin t. Since the events $\mathcal{Q}_{i,t}$ are independent we can write:

$$\Pr[\Phi^t_{ij}] = \Pr[\mathcal{Q}_{i,t} \cap \mathcal{Q}_{j,t}] = \Pr[\mathcal{Q}_{i,t}] \Pr[\mathcal{Q}_{j,t}] = \frac{1}{R}\frac{1}{R} = \frac{1}{R^2} \tag{4.9}$$

Using the Bayes theorem we get:

$$\begin{aligned} \Pr[\Phi^t_{ij} \mid Z_t = 1] =& \frac{\Pr[Z_t = 1 \mid \Phi^t_{ij}] \cdot \Pr[\Phi^t_{ij}]}{\Pr[Z_t = 1]} \\ \overset{(4.9)}{=}& \frac{\Pr[Z_t = 1 \mid \Phi^t_{ij}] \cdot 1/R^2}{\Pr[Z_t = 1]} = \frac{1/R^2}{\Pr[Z_t = 1]} \end{aligned} \tag{4.10}$$

since $\Pr[Z_t = 1 \mid \Phi^t_{ij}] = 1$ (i.e. given that balls i and j fall into bin t, $Z_t = 1$ with probability 1 since there is a collision in that bin). Equation 4.10 dictates that given a collision t, all pairs of balls (i, j) are equally probable to incur that collision t. Thus, if we denote with C^t_{ij} the event where the pair of balls (i, j) incurs a particular collision t (i.e. C^t_{ij} is Φ^t_{ij} given $Z_t = 1$), we can write: $\Pr[C^t_{ij}] = \Pr[C^t_{lm}]$ for any pairs (i, j) and (l, m). The total number of pairs of balls is $\binom{p}{2}$ and therefore:

$$\Pr[C^t_{ij}] = \frac{1}{\binom{p}{2}} = \frac{2}{p(p-1)} \tag{4.11}$$

Finally, if we denote with $\mathcal{I}_i^t$ the event where ball i incurs the collision t (or equivalently a processor i incurs the conflict t) we can write:

$$\mathcal{I}_i^t = \bigcup_{\substack{j=1 \\ j \neq i}}^{p} C_{ij}^t \tag{4.12}$$

where the events C_{ij}^t with i, t fixed and $j \in \{1, ..., p\} - \{i\}$ are mutually exclusive. Therefore:

$$\begin{aligned} \Pr[\mathcal{I}_i^t] &= \Pr[\bigcup_{\substack{j=1 \\ j \neq i}}^{p} C_{ij}^t] = \sum_{\substack{j=1 \\ j \neq i}}^{p} \Pr[C_{ij}^t] \overset{(4.11)}{=} \sum_{\substack{j=1 \\ j \neq i}}^{p} \frac{2}{p(p-1)} \\ &= \frac{2}{p(p-1)} \sum_{\substack{j=1 \\ j \neq i}}^{p} 1 = \frac{2}{p(p-1)} \cdot (p-1) = \frac{2}{p} \end{aligned} \tag{4.13}$$

If we denote with $\mathcal{F}_i$ the number of conflicts that a processor i incurs on the critical path we get:

$$\mathbb{E}[\mathcal{F}_i] = \sum_{t \in \mathcal{T}} \Pr[\mathcal{I}_i^t] \overset{(4.13)}{=} \sum_{t \in \mathcal{T}} \frac{2}{p} = \frac{2}{p} \sum_{t \in \mathcal{T}} 1 = \frac{2}{p} \mathcal{C}_{total}(p) \tag{4.14}$$

Equation 4.14 indicates that the expected number of the conflicts incurred along the critical path is equal to $2 \cdot \mathcal{C}_{total}(p)/p$, which completes the proof. $\square$

4.2.4 Implementation Details

For our implementation we used the portable, high-performance Berkeley UPC compiler [50]. The result is a portable code that can be executed on both shared and distributed memory machines without any change.

Memory Management

We now describe the mechanism that allows us to implement the "aggregating stores" communication optimization. When processor p_i stores S entries to the local-shared stack of p_j, it needs to locate the position in p_j's stack that these entries should go to. Thus, every local-shared stack is associated with its `stack_ptr` pointer that indicates the current position in the local-shared stack. These `stack_ptr` variables are shared and accessible to all processors. Therefore, if processor p_i is about to store S entries to processor p_j, it (a) reads the current value of p_j's `stack_ptr`, called `cur_pos`, (b) increases the value of p_j's `stack_ptr` by S and (c) stores the S entries in p_j's local-shared stack into the locations `cur_pos` $\cdots$ `cur_pos+S-1` with an aggregate transfer. Steps (a) and (b) need to be executed atomically to avoid data hazards, for which we leverage global atomics `atomic_fetchadd()` provided by Berkeley UPC.

Atomic Flag Updates

In order to ensure atomicity in the parallel de Bruijn graph traversal we utilize a lock-free approach that is based on Berkeley UPC atomics. In particular, to atomically read and update a k-mer's `used_flag` (as it is described in Section 4.2.2), we make use of the `compare_and_swap()` global atomic. Thus, if the `used_flag` field is already `USED`, then the atomic returns the value `USED`. Conversely, if the `used_flag` field is `UNUSED` the atomic sets that field to `USED` and returns the value `UNUSED`.

DNA Sequence Compression

Given that the vocabulary in a DNA sequence is the set {`A,C,G,T`} only two-bits per base are required for a binary representation. We thus implement a high-performance compression library that compresses the DNA sequences from text format into a binary format. This approach reduces the memory footprint by $4\times$, while additionally reducing the bandwidth by $4\times$ for communication events that involve k-mers or DNA sequences transfers. Finally we exploit the complementary nature of DNA by not storing explicitly both (redundant) strands; i.e. given one strand of the DNA we can get the other by reversing the former sequence and swapping `A`s with `T`s and `C`s with `G`s. This design decision offers an extra $2\times$ saving in memory requirements (e.g. we can store only the lexicographically smaller representation of a k-mer and its reverse complement), but it complicates a bit the code. For example, given a k-mer sequence we should form its reverse complement, find which one is lexicographically smaller and use that representation in downstream computations.

4.3 Communication-Avoiding de Bruijn Graph Traversal

Although the de Bruijn graph traversal described in the previous section demonstrates high scalability, it inherently suffers from high communication overhead. Because we focus on *de novo* genome assembly (i.e. there is no reference genome), the initial contig construction lacks implicit locality, and it is the goal of the de Bruijn graph traversal to explore connectivity and to find the corresponding connected components. Additionally, due to the nature of DNA, the de Bruijn graph is extremely sparse, as previously discussed. Thus, to explore an additional vertex in the graph and expand a subcontig by one base is it necessary to perform a lookup in the distributed hash table, which with high probability will incur a communication event.

Therefore, given a genome of size G, the asymptotic communication cost of the parallel traversal is $O(G)$ messages, or $O(G/p)$ messages if we measure the number of messages along the critical path. Since the messages have size $O(1)$ (each message is just a single k-mer data structure), the asymptotic bandwidth cost on the critical path is also $O(G/p)$. Note that our parallel algorithm is load balanced and communication along the critical path is

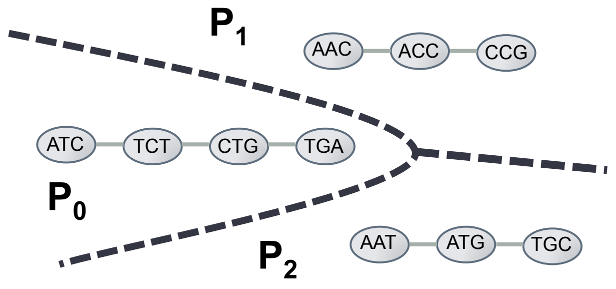


Figure 4.7: A de Bruijn graph of k-mers where $k = 3$. In this graph we can identify three contigs or equivalently three connected components. The dashed lines indicate an optimal partitioning of the graph with $p = 3$. Each processor will work only with local k-mers during the traversal.

decreased as the number of processors is increased, thus allowing our traversal algorithm to strong scale efficiently as the results for the previous scalability analysis dictate.

4.3.1 Insights for improved locality

There are two key insights that underly the development of our communication-avoiding algorithm for the parallel de Bruijn graph traversal:

1. Oracle Partitioning If we were given an *oracle partitioning function* that could accurately predict how the k-mers are placed into contigs, the k-mers could be partitioned in such a way that k-mers ultimately belonging to the same contig would be mapped to the same processor. Thus, during the traversal algorithm a processor would incur *no* communication, since none of the k-mer accesses (lookups in the distributed hash table) will require communication — all the vertices of the de Bruijn graph that build up a particular contig are local to a single processor (local buckets in the distributed hash table). This idea is similar to graph partitioning, which aims to minimize the number of edges between separated components. An example is given in Figure 4.7, which shows three processors and three equally-sized sub-graphs that do not share any edges. By assigning the corresponding k-mers to the appropriate processor, all three processors will conduct local lookups in the hash table during the parallel traversal.

2. Genetic Similarity The nucleotide diversity is similar for given organisms. For example it is estimated that humans differ in only 0.1%–0.4% of DNA base pairs, meaning that the contigs of different humans have a high degree of similarity. This implies that an oracle partitioning derived from one genome will work for others of the same species.

4.3.2 Communication-avoiding parallel algorithm

These insights have enabled us to develop an off-line *oracle partitioning function*. Once the contigs are computed for a given organism, we can apply this oracle partitioning function to *any other* de Bruijn graph (for another member of the same species), to dramatically decrease the communication incurred during the traversal computation. This approach can also be leveraged when searching for the optimal k length for *de novo* assembly. Typically, computational biologists begin the genome assembly process for a given organism with a draft version generated using a reasonable initial k value. Different k lengths are then explored to optimize the quality of the assembly output. Thus we can generate our *oracle partitioning* function during the initial contig generation phase, and use it to significantly reduce communication for subsequent assemblies that explore different k values. Even though the contigs are expected to change for varying k lengths, the new set of contigs will have a high degree of similarity with the first draft assembly, resulting in significant performance improvements. Also, this optimization is applicable in the context of metagenome assembly that is discussed in Chapter 8.

The (off-line) algorithm for generating the oracle partitioning function, `oracle_hash()`, given a set of contigs C and a uniform hash function, `uniform_hash()`, consists of two steps:

1. Iterate over the set of contigs C and assign each contig a processor ID (in a cyclic fashion to ensure load balance) among processors.
2. Iterate within each contig and extract its k-mers. To each one of these k-mers assign the contig's processor ID and store this information in a compact `oracle_hash_vector`. In particular, given a k-mer `A`, we store the corresponding processor ID in the position `uniform_hash(A)` of `oracle_hash_vector`. If there is a collision (i.e. another k-mer of another contig has been already stored in this position of the vector), then k-mer `A` will be stored to a remote processor instead of the correct (local) one. The number of collisions in this `oracle_hash_vector` is approximately the number of communication events that will be incurred during the traversal. Therefore, with a larger `oracle_hash_vector` we can decrease the number of collisions and consequently we can decrease the communication volume. Essentially we can trade off memory requirements and the number of collisions in the `oracle_hash_vector`, thus increasing or decreasing communication overhead according to the size of the available aggregate memory. Note that the algorithm for generating the `oracle_hash_vector` can be trivially parallelized. Nevertheless, since the construction of the `oracle_hash_vector` is an offline process and has to be completed only once, it does not lie on the critical path of the pipeline's execution.

The `oracle_hash()` values are computed during the graph construction and traversal for a given `oracle_hash_vector` as follows:

1. The processors load the `oracle_hash_vector` in their local memory – note that the `oracle_hash_vector` is replicated across processors. Alternatively we can replicate

the `oracle_hash_vector` on a node basis in order to decrease the per thread memory requirements.

2. When calling `oracle_hash(A)` at the application level in respect to a k-mer `A`, the value of `uniform_hash(A)` is computed. Next we lookup in the `oracle_hash_vector` that is local to the processor and retrieve the corresponding processor ID (p_i). Based on the value of p_i, we subsequently reshuffle the original `uniform_hash(A)` value. In particular, if `uniform_hash(A)` was about to be mapped in location (bucket) b of processor p_j (assuming a cyclic distribution of buckets to processors), the return value of `oracle_hash(A)` is adjusted such that it is mapped at location (bucket) b of processor p_i. Observe that uniformity in the distribution of buckets is preserved since all the hash values are shifted in a uniform way to the appropriate buckets. Moreover, the vast majority of the contig's k-mers that will be looked up after k-mer `A` during the traversal phase, are located in the same processor p_i by construction of the `oracle_hash_vector`. Therefore, if the processors select traversal seeds from local buckets, they will be mostly performing local accesses in the hash table when traversing the de Bruijn graph.

A refinement for practical considerations (e.g. SMP clusters), is working with node IDs instead of processor IDs. In this setting, the algorithm avoids the off-node communication while performing intra-node accesses when generating the contigs, which are orders of magnitude cheaper than the off-node communication overhead.

4.4 Experimental Results

We experimented again on the human and the wheat datasets described in Section 3.4.1. The experimental platform is again the Edison supercomputer. Our experiments sought to understand the strong scaling performance for these two data sets that differ by a factor of 6× in size.

4.4.1 Parallel De Bruijn Graph Construction and Traversal

Figure 4.8 exhibits the strong scaling behavior of our de Bruijn graph construction and traversal using the human data set. The experiments presented in this section do not utilize the communication-avoiding optimization. We were not able to run this step for less than 480 cores, due to memory limitations. Our implementation shows efficient scaling for both constructing and traversing the de Bruijn graph, attaining a 12.24× relative speedup when increasing concurrency from 480 to 15,360 cores. We highlight that due to our optimized parallelization approach, our 15,360 core execution completes the contig generation for the human genome in $\sim$ 20 seconds, compared with the original Meraculous serial code that required $\sim$ 44 hours on a 128 GB shared memory node.

Figure 4.9 presents the strong scaling results for the wheat data set. The starting point for the graph is 960 processors, since the memory requirements are significantly larger than

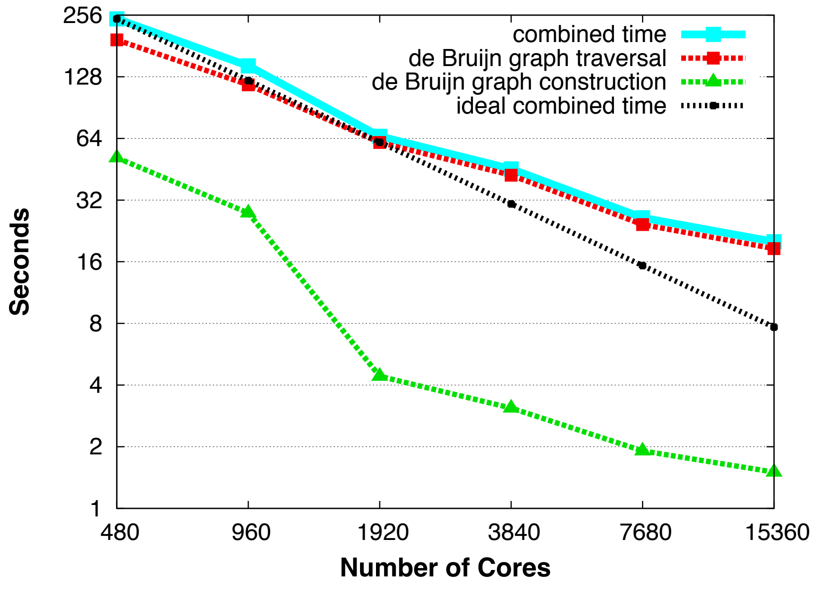


Figure 4.8: Strong scaling of the contig generation on Edison for the human dataset (without the communication-avoiding optimization). Both axes are in log scale. Combined time corresponds to the sum of de Bruijn graph construction and traversal times.

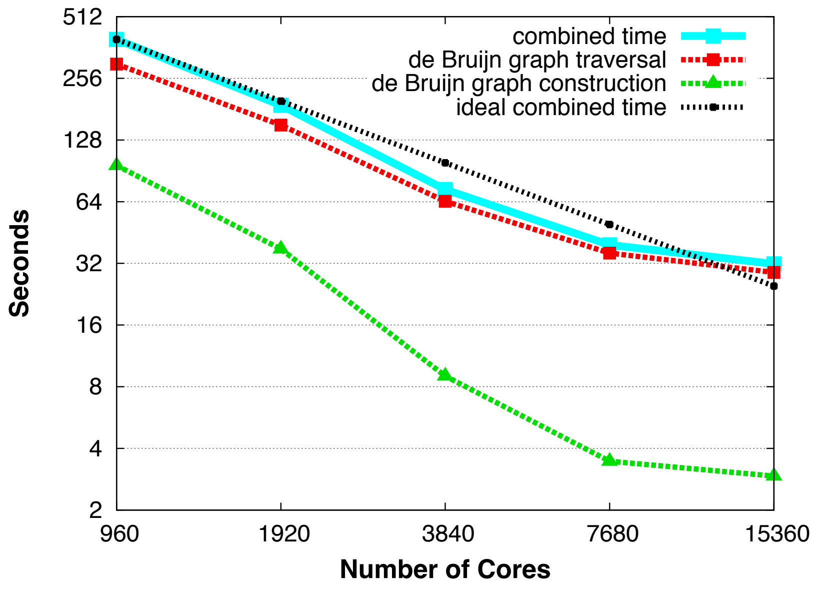
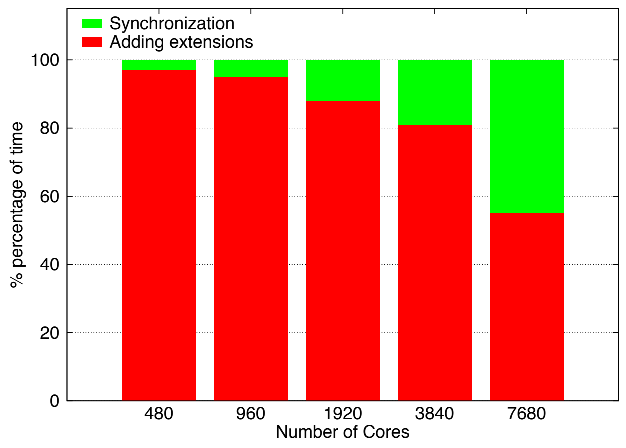
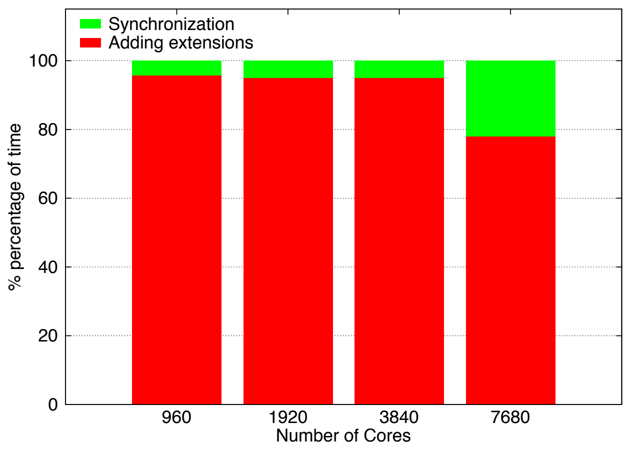


Figure 4.9: Strong scaling of the contig generation on Edison for the wheat dataset (without the communication-avoiding optimization).

the human dataset. We achieve 12.5× relative speedup when increasing concurrency from 960 to 15,360 cores. For this grand-challenge genome, our implementation required only



(a)



(b)

Figure 4.10: Time breakdown of the de Bruijn graph traversal for (a) the human genome and (b) the wheat genome.

$\sim$ 32 seconds on 15,360 cores, whereas single-node memory constraints prevent the original Meraculous version from running this problem. We extrapolate that if enough memory was available, the original Meraculous code would have required 121 hours.

Additionally, for each strong scaling experiment we measured the fraction of time spent to add forward/backward extensions at the subcontigs, because this is the useful workload and the fraction of time spent in the synchronization protocol along the critical path. Figure 4.10 shows this performance breakdown. We observe that although the relative time spent in synchronization mode increases for higher concurrency levels, the majority of the time is always consumed in doing useful work. This empirical behavior validates our analysis in Section 4.2.3 and demonstrates that our synchronization scheme is lightweight, enabling our parallel algorithm to scale up to tens of thousands of cores.

4.4.2 Communication-Avoiding de Bruijn Graph Traversal

To evaluate the effectiveness of our communication-avoiding algorithm we investigate performance results on the human genome dataset on Edison using two concurrencies: 480 and 1,920 cores. Table 4.1 presents the speedup achieved by our communication-avoiding algorithm (columns "oracle-1" and "oracle-4") over the basic version without an oracle hash function (column "no-Oracle"). The case labeled "oracle-1" corresponds to an oracle hash function with a per-thread memory requirement of 115 MB, while the "oracle-4" hash functions requires four times more memory, i.e. 461 MB per-thread. Recall that an increase of dedicated memory for the oracle hash function corresponds to more effective communication-avoidance. At 480 cores the communication-avoiding algorithms yields a significant speedup up to 2.8$\times$ over the basic algorithm, while at 1,920 cores we achieve an improvement in performance up to 1.9$\times$.

Cores	**Graph traversal time (sec)**			**Speedup**	
	no-Oracle	oracle-1	oracle-4	oracle-1	oracle-4
480	145.8	105.8	52.1	1.4×	2.8×
1,920	46.3	35.9	24.8	1.3×	1.9×

Table 4.1: Speedup of communication-avoiding parallel de Bruijn graph traversal vs. the basic (no Oracle) algorithm.

Cores	**Off-node communication time (% of total communication time)**			**% reduction in off-node communication time**	
	no-Oracle	oracle-1	oracle-4	oracle-1	oracle-4
480	92.8 %	54.6 %	22.8 %	41.2 %	75.5 %
1,920	97.2 %	54.5 %	23.0 %	44.0 %	76.3 %

Table 4.2: Reduction in communication time via oracle hash functions.

The data presented in Table 4.2 show that the performance improvements are related to a reduction in communication. As expected, the basic algorithm performs mostly off-node communication during the traversal. In particular, 92.8% of the lookups result in off-node communication at 480 cores, and 97.2% of the lookups yield off-node communication at 1,920 cores. By contrast, even a lightweight oracle hash function "oracle-1" reduces the off-node communication by 41.2% and 44%, at 480 and 1,920 cores, respectively. By allocating more memory for the oracle hash function ("oracle-4") we can further decrease the off-node communication, by 75.5% and 76.3%.

4.5 Conclusion

In this Chapter we presented parallel algorithms for the two phases of contig generation, namely de Bruijn graph construction and traversal. First, we described how to optimize the de Bruijn graph construction by aggregating dynamically messages. Then, we detailed our parallel de Bruijn graph traversal algorithm tailored for high diameter graphs, where traditional Breadth First Search (BFS) approaches fail to scale. Our algorithm employs massive parallelism and a provably lightweight synchronization scheme. We further optimized the de Bruijn graph traversal by developing a communication-avoiding technique inspired by graph partitioning and genetic similarity. Human re-sequencing projects, optimization of the single genome assembly and metagenome assembly can take advantage of this communication-avoiding optimization. Finally, we presented distributed memory results up to 15K cores on the Edison supercomputer and demonstrated that our algorithms scale efficiently despite their irregular nature. As a result, our approach overcomes the limitations of specialized big shared memory machines and allows the contig generation to be completed in seconds instead of days.

Chapter 5

Parallel Sequence Alignment

Aligning a set of query sequences to a set of target sequences is an important task in bioinformatics. For example, the human genome is often analyzed by aligning each new individual's genome against a reference. In our assembly pipeline (see Figure 2.2(a)), we will use alignment to extend the length of the contigs by referring to the original reads – among other things, this will help to resolve branches in the de Bruijn graph and disambiguate repeated sequences that did not have unique extensions. The alignment of two sequences does not require a perfect match, but allows some number of mismatched characters, insertions and deletions. The bioinformatics community has therefore developed several approaches for parallelizing the alignment of multiple reads (*queries*) to a set of reference sequences (*targets* or *contigs* in the Meraculous pipeline).

A class of sequence alignment methods include the seed–and–extend algorithms (e.g. BLAST [3]). The seeds are sequences of fixed length that can be extracted from the queries or the target sequences. In the seed–and–extend paradigm, we extract first all possible seeds that can be found in the reference sequences and store them in a hash table called *seed index*. By searching a seed in the seed index we get as a result all the reference sequences that contain this seed. In this way, by looking up seeds extracted from the queries we can find in constant time potential pairs of queries and references to be aligned, with the assumption being that all alignable pairs will have at least one common (identical) seed. Finally, an extension algorithm is applied to the alignable pairs and we find detailed alignments of queries with references.

In some applications of this methodology, the reference genome is known a priori, thus allowing an off-line seed index construction that can then be exploited for multiple read data sets. This scenario allows for straightforward parallelization, where the seed index is replicated across a set of computational nodes, which can then independently and concurrently align their subset of the reads. Indeed, there are existing frameworks (e.g. pMap [88]) that automate the process of (1) index replication, (2) distribution of reads across nodes and (3) local alignment computation.

However, this approach can suffer from two major limitations. First, the seed index of very large genomes (e.g. wheat [20], pine [110]) and metagenomes may exceed the memory capacity of a singe node, thereby preventing the use of a simple seed index replication scheme. More significantly, there are important applications such as de novo genome assembly where the reference sequence is not known ahead of time, thus obviating the off-line approach. Therefore, parallel de novo genome assemblers rely on efficient aligner algorithms, where the seed index construction must be efficiently parallelized and distributed to allow high concurrency solutions for grand-challenge genomes and metagenomes. In the Meraculous pipeline, seeds from reads and contigs are extracted using the same sliding window of k-mers that is used in contig generation. In this chapter we describe the alignment process as a standalone algorithm with two sets of sequence data, the target sequences (reference) and the query sequences (reads), since alignment is useful in other genomic analyses besides assembly. In our pipeline the target is the set of contigs, since we have high confidence that those are correct, and the queries are the original reads. The reads have information about how the contigs may connect to one another, and the alignment phase will help to capture that.

The parallel alignment algorithm presented in this chapter[1] is called *merAligner* [34]. All the components of merAligner are fully parallelized from end to end, including the I/O and the seed index construction. The seed index implementation presented in this chapter leverages dynamic message aggregation and software caching to minimize communication overheads. Also, merAligner preprocesses the reference sequences and performs exact seed matching with minimal communication and computation without sacrificing accuracy. The performance results of merAligner show close-to-ideal scaling (with 0.7 - 0.78 parallel efficiency) up to 15K cores on NERSC's Edison Cray XC30 supercomputer, using real datasets from the human and the grand-challenge wheat genomes. Finally, we compare merAligner with existing alignment solutions, showing the significant advantage of our end-to-end parallel approach. Overall, this chapter shows that efficient utilization of distributed memory architectures enables effective parallelization of sequence alignment in terms of both high scalability and reduced per-node memory requirements.

5.1 The merAligner Parallel Algorithm

Algorithm 2 describes the parallel seed–and–extend algorithm we employ to align a set of query sequences to a set of target (reference) sequences. We emphasize here that the pseudocode of Algorithm 2 assumes a Single Program, Multiple Data (SPMD) parallel model where all processors execute the same program. The algorithm outputs detailed alignments of queries with targets. In the context of this chapter, an alignment of two sequences is a way of arranging them in order to identify regions of similarity. In this arrangement, the sequences can have exact matches in the corresponding positions, mismatches or inserted gaps. Also, an alignment is characterized by a score and typically an exact character match has a positive contribution in the score while mismatches and inserted gaps have a negative

[1]The material presented in this chapter was first published in the paper "merAligner: A Fully Parallel Sequence Aligner" [34]

Algorithm 2 Parallel sequence alignment

```
1: Input: A set of queries and a set of targets, both sets of DNA sequences
2: Output: Set of alignments of queries with targets
3: $mySetOfAlignments \leftarrow \emptyset$
4: for all processors $p_i$ in parallel do
5:     $myTargetSeqs \leftarrow$ ReadTargetSequences($targets$)
6:     $mySeedsInTargets \leftarrow$ ExtractSeeds($myTargetSeqs$)
7:     $globalSeedIndex \leftarrow$ AllCreateGlobalSeedIndex($mySeedsInTargets$)
8:     $myQuerySequences \leftarrow$ ReadQuerySequences($queries$)
9:     for each query sequence $q \in myQuerySequences$ do
10:        for each seed $s \in q$ do
11:            $myCandidateTargets \leftarrow$ LookUp($globalSeedIndex, s$)
12:            for each target $t \in myCandidateTargets$ do
13:                $myNewAlignment \leftarrow$ SmithWaterman($t, q$)
14:                $mySetOfAlignments \leftarrow$ AddToSet($myNewAlignment, mySetOfAlignments$)
```

impact on the score. An optimal alignment represents an arrangement of the two sequences such that the score is maximized given the match reward and the mismatch and gap penalties. Figure 5.1 illustrates an example of how two sequences can be aligned assuming that the match, mismatch and gap scores are 1, -2 and -5 respectively. In subsection 5.1.4 we describe the Smith-Waterman [99] dynamic programming algorithm that is used in order to find optimal alignments.

5.1.1 Extracting Seeds from Target Sequences

First, each processor p_i reads a distinct portion of the target sequences (line 5) and stores them in shared memory such that any other processor can access them. Every target sequence of length L contains $L - k + 1$ distinct seeds of length k. The first bases $1 \ldots k$ of a target form the first seed, the bases $2 \ldots k + 1$ form the second seed, etc. We extract seeds from the target sequences and associate with every seed the target from which it was extracted (line 6) – we also keep track of the exact offset of the seed in the target. Note that a given seed s might appear in two or more target sequences.

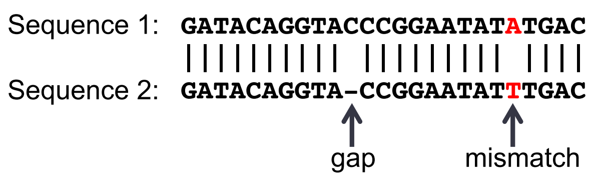

Sequence 1: GATACAGGTACCCGGAATATATGAC
Sequence 2: GATACAGGTA-CCGGAATATTTGAC
gap
mismatch

Figure 5.1: Detailed alignment of two DNA sequences. Assuming that the match, mismatch and gap scores are 1, -2 and -5 respectively, the total score of this alignment is 16.

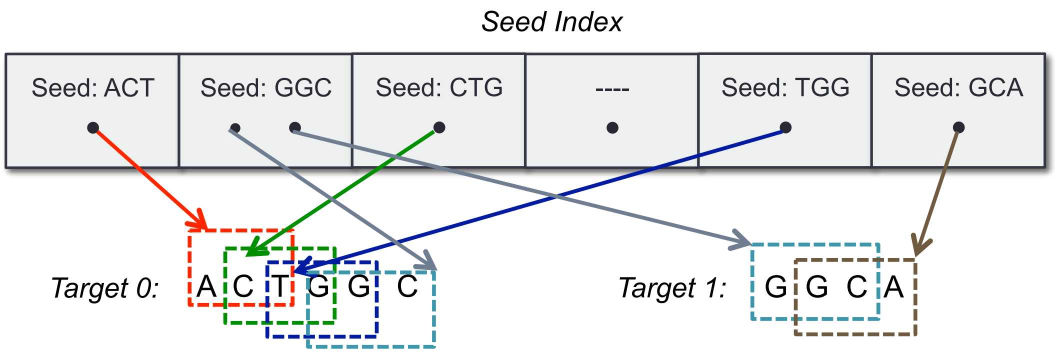


Figure 5.2: An example of a seed index data structure that indexes two target sequences. Note that the seed index is distributed and stored in global shared memory. Also, the target sequences are stored in global shared memory such that any processor can access them. Here the seed `GGC` is extracted from both target sequences, thus the value in the corresponding hash table entry is a list of pointers to the corresponding sequences. For simplicity we do not show here additional stored information, such as the seed's offsets in the targets.

5.1.2 Indexing Target Sequences

Once the seeds are extracted from the target sequences, they are stored in a global hash table, henceforth referred to as the *seed index* (line 7), where the key is a *seed* and the value is a pointer to the target sequence from which this *seed* has been extracted. If a *seed* is extracted from multiple target sequences, its value in the hash table is a list of pointers to those targets. The seed index is distributed and stored in global shared memory such that any processor can access and lookup any seed. Essentially the seed index data structure provides a mapping from seeds to target sequences (see Figure 5.2).

5.1.3 Locating Query-to-Target Candidate Alignments

Given a seed s from a query sequence q and an index $globalSeedIndex$, we perform a lookup and locate the candidate target sequences that have $length(s)$ consecutive bases matching with q (line 11). Thus, each one of the query-to-target candidate alignments can be located in $O(1)$ time (see Figure 5.3).

5.1.4 Identifying Alignments via Smith-Waterman

Finally, after locating a candidate target sequence t that has $length(s)$ consecutive bases matching with a query sequence q (where s is a common seed in both sequences), the Smith-Waterman [99] algorithm is executed with input the sequences t and q in order to perform local sequence alignment (line 13). The Smith-Waterman algorithm is a general local alignment method based on dynamic programming. The algorithm compares segments of all possible lengths and optimizes the similarity measure. Therefore, the core of this algorithm

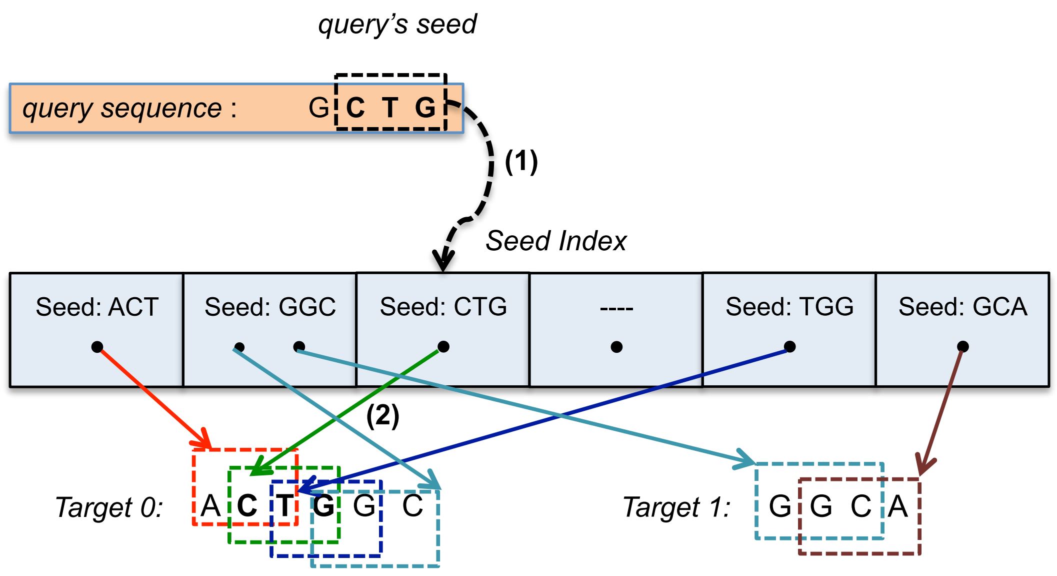


Figure 5.3: Locating query-to-target candidate alignments. First the processor extracts a seed from the query sequence (`CTG` seed). Next the processor looks up the distributed seed index (arrow 1) and finds that a candidate target sequence is Target 0 (arrow 2). Finally, the Smith-Waterman algorithm is executed using as inputs the query and the Target 0 sequences.

is the computation of a similarity matrix H. In this formulation q, t are string sequences over the alphabet {`A`,`C`,`G`,`T`}, q_i is the i-th character of a string q, t_j is the j-th character of a string t and w is a measure of weight for two characters. By denoting:

- $w(q_i, t_j) = match\_score$ if $q_i = t_j$
- $w(q_i, t_j) = mismatch\_score$ if $q_i \neq t_j$
- $-$ is the gap symbol and stands for deletion or insertion
- $w(q_i, -) = w(-, t_j) = gap\_score$
- $H(i, j)$ is the maximum similarity-score between a suffix of $q[1...i]$ and a suffix of $t[1...j]$

the similarity matrix H is built from the following recursive expressions (given the integer scores $match\_score$, $mismatch\_score$ and gap_score where typically $match\_score$ is positive and $mismatch\_score$ and gap_score are negative):

$$H(i, 0) = 0, \; 0 \leq i \leq length(q)$$

$$H(0, j) = 0, \; 0 \leq j \leq length(t)$$

$$H(i,j) = max \begin{cases} 0 & \\ H(i-1,j-1) + w(q_i, t_j) & Match/Mismatch \\ H(i-1,j) + w(q_i, -) & Deletion \\ H(i,j-1) + w(-, t_j) & Insertion \end{cases}$$

with $1 \leq i \leq length(q), 1 \leq j \leq length(t)$.

Once the computation of matrix H is completed, the alignment is reconstructed as follows: Starting with the last value $H(length(q), length(t))$ we find the (i,j) entry that was used to arrive at the current location in the matrix H. A diagonal jump implies there is an alignment (either a match or a mismatch). A top-down jump implies there is a deletion. A left-right jump implies there is an insertion. We can recursively apply this idea to find the optimal alignment arrangement of q and t. The Smith-Waterman algorithm is fairly demanding in time since the similarity matrix H takes $O(length(q) \times length(t))$ time to be computed.

However, two sequences q and t that align well will generate a path through the Smith-Waterman alignment matrix that will almost run diagonally from the start point $(0,0)$ to the end point $(length(q), length(t))$. Only insertions and deletions will cause a horizontal or vertical shift on the diagonal alignment path. We can exploit this property and calculate only a part of the similarity matrix, typically a band of width β around the diagonal. This algorithm is called *banded* Smith-Waterman and has time complexity $O(\beta \cdot \min(length(q), length(t)))$. In our application we know that the sequences q and t are quite similar since they have a common seed s and as a result we can leverage the banded version of Smith-Waterman.

5.2 Distributed Seed Index Optimizations

In this section we detail the optimizations we employed regarding the distributed seed index.

5.2.1 Distributed Seed Index Construction

The computational task of the seed index construction is to insert all the seeds from the target sequences into a distributed hash table (seed index) that can be globally accessed for later use. We recognize this computational pattern as the Use Case 1 (GUO) of the distributed hash tables (see subsection 2.3.2), therefore we can mitigate the communication and synchronization overheads by leveraging dynamic message aggregation. This is exactly the same optimization described in subsection 4.1.1, where it is used for the construction of the distributed de Bruijn graph.

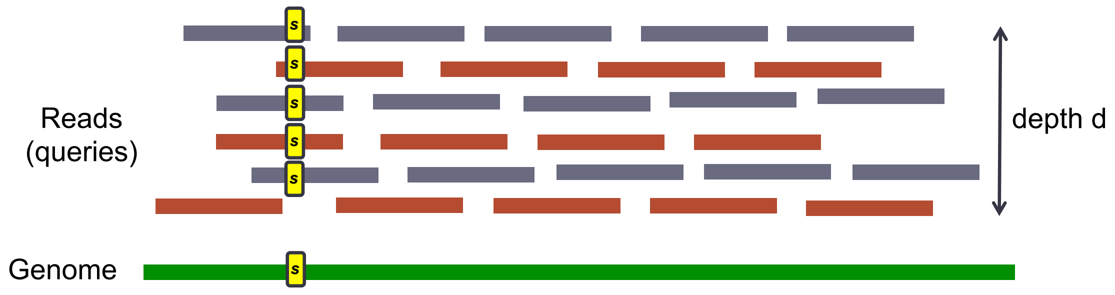


Figure 5.4: A genome sampled at some depth of coverage d.

5.2.2 Software Caching Schemes

The nature of the alignment problem enables data reuse in both the seeds and target sequences, allowing us to exploit this insight for a more efficient implementation. High throughput sequencing allows genomes to be sampled redundantly at a depth d, as visualized in Figure 5.4. Let k be the length of a seed s and L be the read length. Any seed s of the genome (yellow region) is expected to be found $f = d \cdot (1-(k-1)/L)$ times in the read data set (f is the mean of the Poisson distribution of key-frequencies [68]), thus resulting in f lookups for that seed within the distributed seed index. Additionally, targets are in general sequences that are significantly longer than the reads. Thus, multiple reads are expected to be aligned with the same target and a given target t is expected to be reused multiple times in the seed extension procedure.

The parallel alignment algorithm makes no writes/updates in the distributed seed index and the distributed data structure that stores the target sequences after their construction phase; it just uses them for lookups/reads. We recognize this computational pattern as the Use Case 3 (GRO) of the distributed hash tables (see subsection 2.3.2) and given the potential for data reuse, we developed a software cache architecture to reduce communication overhead as shown in Figure 5.5. In UPC, the address space of every node is logically divided into *private memory* and *shared memory*. The private memory is thread local and can only be accessed by the UPC thread (which maps to a processor in our case) to which it has affinity. On the other hand, a location of the shared memory can be accessed by *any* UPC thread in the system. It is much faster to access locally stored data than to access shared memory residing on a remote node. Thus, on every node, a portion of the shared memory is dedicated for software caches that can store either remote parts of the distributed seed index (seed index cache) or target sequences owned by remote nodes (target cache). In Figure 5.5 consider Node i which has stored in its seed index cache a seed s (yellow block) that belongs to the part of the distributed seed index local to Node j. Any lookup for the seed s by processors of Node i will be served by the seed index cache resulting in much faster lookup time than accessing the original yellow block on the remote Node j. Similarly, consider a target sequence t (red block) which has been stored to the target cache of Node i. Processors of Node i that need to align a query with respect to t will fetch t from the target cache and

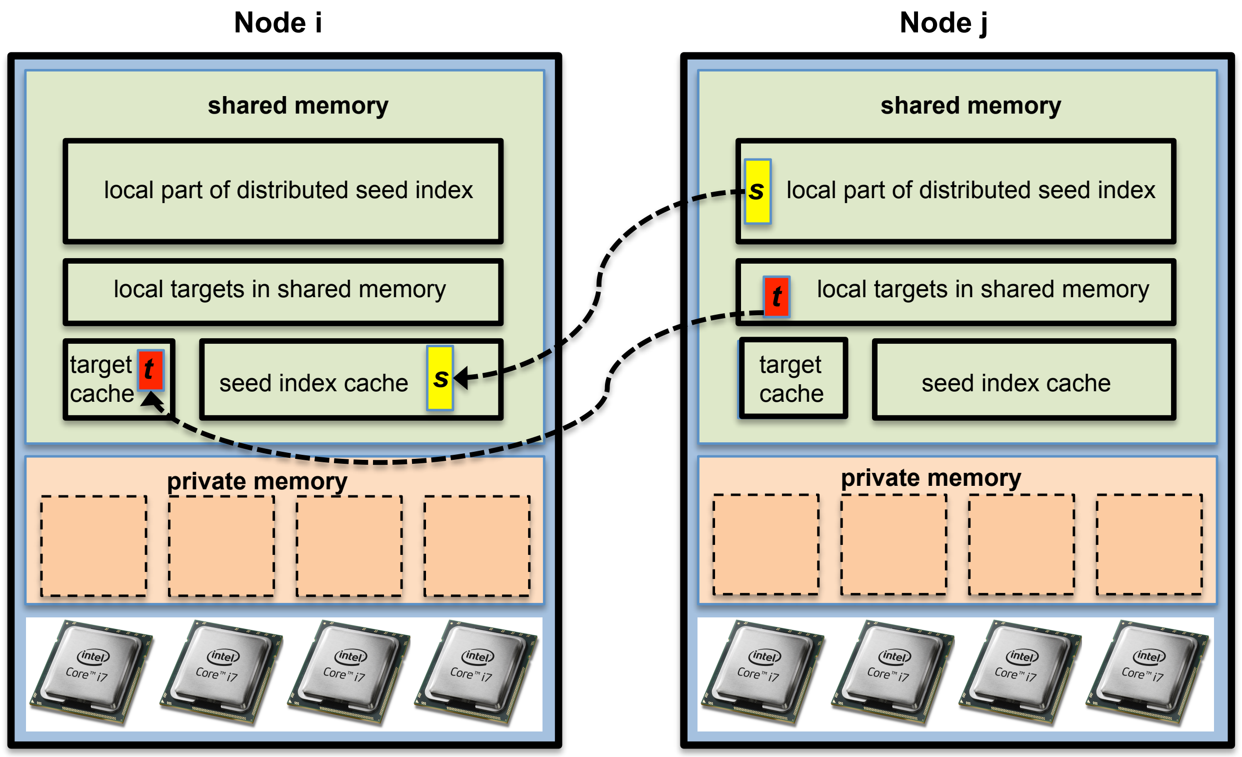

Node i
Node j
shared memory
local part of distributed seed index
local targets in shared memory
target cache
t
seed index cache
s
private memory
shared memory
s
local part of distributed seed index
t
local targets in shared memory
target cache
seed index cache
private memory

Figure 5.5: Software cache architecture for the distributed seed index and the target sequences. Node i has stored in its seed index cache a seed s (yellow block) from Node j. Any lookup for the seed s by processors of Node i will be served by the seed index cache. Similarly, Node i has stored in its target cache a sequence t (red block) from Node j. Any processor of Node i that needs to align a query with respect to t will fetch t from the target cache.

thus avoid the expensive off-node communication.

The expected seed data reuse naturally depends on the seed distribution among processors. As discussed later in Subsection 5.3.2, for load balancing reasons reads are assigned to processors in a uniformly random fashion. Consider a parallel system with p total processors, with ppn processors per node and a seed s with frequency f in the read data set residing on node i. Following the reasoning in the previous paragraph, there are $f - 1$ additional occurrences of that seed s in the read data set or equivalently there are $f - 1$ locations in the reads that include that seed.

We can then ask the question: What is the probability that at least one such read is assigned to node i? This problem can be reduced to the well known "bins and balls" experiment. In this case, given $f - 1$ balls (remaining occurrences of the seed s) and $m = p/ppn$ bins (nodes), we toss the balls (reads) uniformly at random — the probability that *at least* one of these balls falls in bin i (node i) is $1-(1-1/m)^{f-1}$. Therefore, with probability

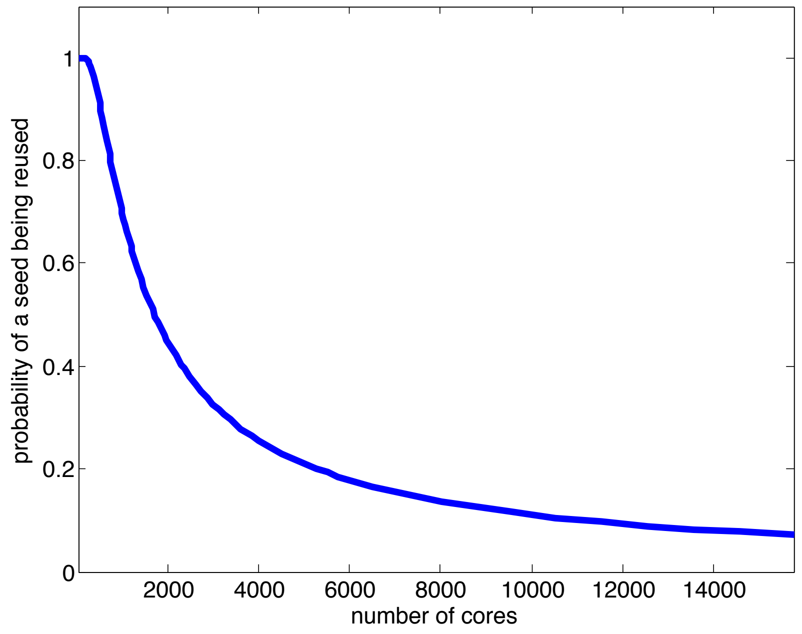


Figure 5.6: Probability of any seed being reused as a function of cores. We have picked values of $d = 100$, $L = 100$, $k = 51$, $f = 50$ and *ppn*=24.

$1 - (1 - 1/m)^{f-1}$ our approach will perform *at least* one seed index lookup of s resulting in a software cache hit, since there are at least two occurrences of that seed in the same node i. In order to assess the limits of this optimization, consider the case of a read data set with $d = 100$, $L = 100$, $k = 51$, $f = 50$ and a system with *ppn* $= 24$ cores per node. Figure 5.6 shows the probability of any seed being reused at least one time given the previous values of f and *ppn*. Note, however, that this is the behavior in the ideal case of "infinite" cache. In practice, we dedicate a fraction of the nodes' memory for software caching, and tradeoff memory for increased data reuse. For typical experimental values of d and *ppn*, we expect a significant benefit from our cache optimization strategy, as demonstrated in the experimental results of Section 5.5.

5.3 Alignment Optimizations

We now discuss our alignment optimizations and theoretical proofs of expected behavior.

5.3.1 Optimizing exact read matches

Here we devise a method to preprocess the target sequences to identify properties enabling exact sequence matching with minimal communication. The property we describe is based on the Lemma 1.

Lemma 1. *Let k be the length of the seeds, q be a query sequence and t be a target sequence where **all** the seeds extracted from t are uniquely located in t (i.e. this seed cannot be found in any other target sequence). Assume that s is a subsequence in q with $length(s) \geq k$ and t is a candidate target to be aligned with q that also includes the subsequence s. Then, q is **uniquely** aligned to t with respect to the subsequence s, in essence there is no other target t' that matches with q in the subsequence s.*

Proof. Since q matches with t in $length(s)$ bases, then all $length(s) - k + 1$ seeds in s belong to both q and t. Since all seeds in t are *uniquely* located in t we conclude that also these $length(s) - k + 1$ seeds are uniquely located in t and therefore there are no other targets that include those seeds. Consequently, there are no other targets that include the subsequence s. □

Consider a subsequence $s \equiv q$, and assume that the first candidate target t_0 is to be aligned with q and it is known that **all** the seeds extracted from t_0 are uniquely located in t_0 (we detail in the subsequent paragraph how to identify such a property). With a fast check we can determine if q and t_0 match in exactly $length(s)$ bases. Given this scenario, then via Lemma 1 with $s \equiv q$ it holds that q is **uniquely** aligned to t_0. Thus it is not necessary to look for more candidate targets and additional seed lookups in the distributed seed index can be avoided. It is thus assured that all possible alignments of q are found (to the set of the targets) by simply performing a single seed lookup — thereby only requiring minimal communication. Further speedups can also be achieved by recognizing that a seed extension algorithm is not necessary in this case, instead a simple and fast string comparison between q and the appropriate location of t_0 can be executed.

We now explain how to identify, efficiently for all target sequences, whether the seeds extracted from t_0 are uniquely located in t_0. During the distributed seed index construction described in Subsection 5.2.1, when a processor adds the received seeds in its local buckets of the hash table, it counts the number of occurrences of each seed — a cheap and local operation. We additionally associate a boolean `single_copy_seeds` flag that is initialized as `true` for all targets. After inserting the seeds into the seed index, a processor p_i can visit all the local seeds and if the count of an encountered seed s' is greater than 1, p_i sets the flags `single_copy_seeds` of the targets that s' was extracted from as `false`. This indicates that those targets do **not** have seeds uniquely located in them. At the end of this step, all the remaining targets with `single_copy_seeds` set to true are guaranteed to have **all** their seeds uniquely located in them.

To maximize the impact of this optimization, we add an additional strategy. Given the seed length k, the longer a target sequence t is, the more probable it is that t contains at least one seed that is not uniquely located in t, thus negating the potential of leveraging the described lookup optimization (even if some reads uniquely match to t). Now consider the case where a target sequence t' has all but one seed a uniquely located in t'. If we fragment t' in two equal-length subsequences t'_1 and t'_2 (that overlap to some degree but have disjoint sets of seeds), then the non-uniquely located seed a in t' should be found (by construction) in either t'_1 or t'_2. Thus the subsequence not containing the seed a consists of uniquely located seeds, thereby enabling our described optimization.

The same reasoning can be applied recursively to address the general case where a target t' contains multiple non-uniquely located seeds. The idea is to fragment the sequence t' into m equal-length subsequences $t'_1, t'_2, ..., t'_m$ that overlap to some degree — however the subsequences have disjoint sets of seeds and the union of their sets of seeds is exactly the set of seeds in the original sequence t'. This approach increases the probability of applying the previous optimization. Note that some additional information must be stored for each one of the subsequences $t'_1, t'_2, ..., t'_m$, to allow quick locating of these subsequences later in the alignment.

5.3.2 Load Balancing

Load balancing the queries might initially seem trivial: given n queries and p processors each processor should process n/p queries. Unfortunately, queries may differ in their processing requirements. For instance, consider a query q' that perfectly aligns with a single target sequence. Let $t_{extractSeed}$ be the required time to extract a seed from a query, $t_{lookupSeed}$ the time to lookup a seed in the seed index, $t_{fetchTarget}$ the time to fetch a target sequence, and $t_{memcmp()}$ the time to perform a `memcmp()` operation on $length(q')$ bytes. Then, the time Algorithm 2 takes (after applying the previous optimization) to process q' is $t_{q'} = t_{extractSeed} + t_{lookupSeed} + t_{fetchTarget} + t_{memcmp()}$. On the other hand, consider a query q'' that can be aligned with C targets. Assume that t_{SW} is the time to execute the Smith-Waterman algorithm. Then, processing q'' takes $t_{q''} = L \cdot (t_{extractSeed} + t_{lookupSeed}) + C \cdot (t_{fetchTarget} + t_{SW})$ time, where $L = length(q'') - length(seed) + 1$. Given $t_{memcmp()} \leq t_{SW}$, it must hold that $t_{q''} \geq min(C, L) \cdot t_{q'}$, thus the processing times of two queries can vary significantly.

Assume that the n queries can be divided in two categories: "fast" and "slow" (depending on their required processing time). The goal is to evenly distribute the slow queries to the available p processors. However, because it is unknown a priori if a query is fast or slow, we implement the following load balancing strategy. Before executing Algorithm 2 the order of the queries is randomly permuted in the input file and each processor is assigned a chunk of n/p consecutive queries from the corresponding file. As proven in Theorem 1, if there are h "slow" queries, p available processors and $p \log p \ll h \leq p$ polylog$(p)^2$, then with high probability the load imbalance (difference of maximum "slow" load from the average "slow" load h/p) is at most: $2\sqrt{2\frac{h}{p}\log p}$.

Theorem 1. *Let h be the number of "slow" queries and p be the number of available processors and assume that $p \log p \ll h \leq p$ polylog(p). After assigning the h queries randomly to the p processors (or equivalently randomly permuting the order of the queries in the input file) then with high probability the load imbalance (difference of maximum "slow" load from the average "slow" load h/p) is at most: $2\sqrt{2\frac{h}{p}\log p}$.*

[2] polylog(p) is some polynomial in log(p)

Proof. We formulate the process of randomly permuting the order of the queries in the input file as the uniformly random tossing of h balls into p bins. Let M be the random variable that counts the maximum number of balls ("slow" queries) in any bin. It therefore holds that $Pr[M \leq k] = 1 - o(1)$ where $k = \frac{h}{p} + 2\sqrt{2\frac{h}{p}\log p}$ [89], i.e. with high probability the load imbalance (difference of maximum "slow" load from the average "slow" load h/p) is at most: $2\sqrt{2\frac{h}{p}\log p}$. $\square$

5.3.3 Restricting the Maximum Alignments per Seed

Even after applying the described load balancing scheme, there may be a few seeds that can be aligned with too many targets, causing a high processing time for the corresponding queries. Additionally, finding those numerous alignments may not be relevant to many genome alignment applications. Thus, a threshold can be set for the maximum number of alignments per seed, after which the candidate alignment queries are stopped. This threshold determines the sensitivity of our aligner and it can be used to trade off accuracy for speed when appropriate.

5.4 Additional Optimizations

We now describe the additional set of I/O, SIMD, and compression optimizations utilized in our work.

5.4.1 Parallel I/O

The input read sequences are stored in the standard FASTQ format and we leverage our parallel FASTQ reader described in Section 3.3.

5.4.2 SIMD Optimized Striped Smith-Waterman

MerAligner spends a significant portion of its runtime using the banded Smith-Waterman (SW) algorithm for seed extension. Due to the critical role of SW, many efforts have been made to accelerate it by taking the advantages of special hardware SIMD (Single Instruction Multiple Data) instructions. In this work we incorporate such an implementation from the Striped Smith-Waterman (SSW) library [108] which has been shown to be orders of magnitude faster than reference implementations of SW in C.

5.4.3 DNA Sequence Compression

Given the {`A,C,G,T`} vocabulary of a DNA sequence, only two-bits per base are required for binary representations. We thus use a high-performance compression library that transforms

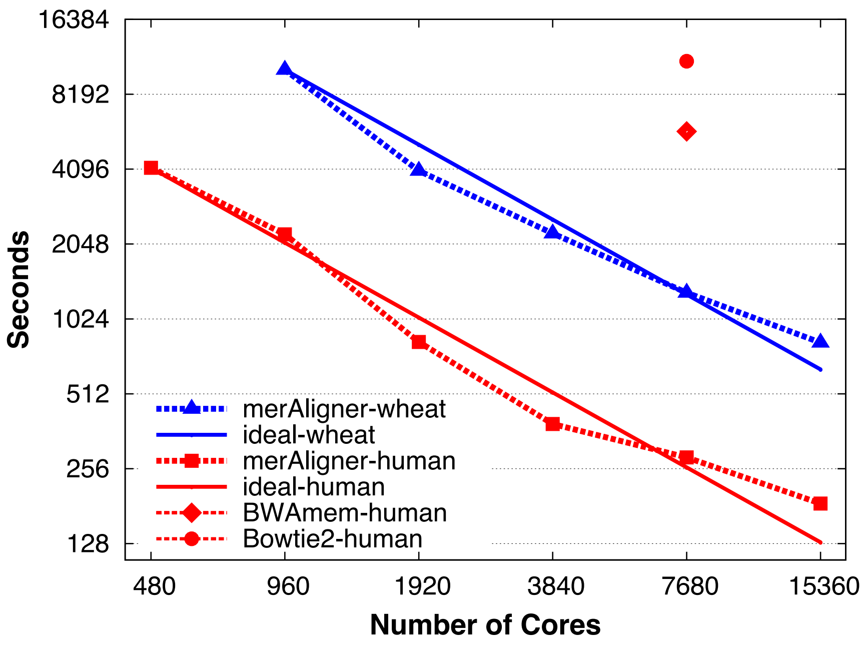


Figure 5.7: End-to-end strong scaling of merAligner on Cray XC30 for the human and the wheat genome. The plotted curves exhibit the performance of merAligner while the single data points show the performance of BWA-mem and Bowtie2 used in the pMap parallel framework.

the DNA sequences from text format into a binary format [35]. This approach reduces the memory footprint by 4×, while also reducing the bandwidth by 4× for communication events that involve seeds or DNA sequence transfers.

5.5 Experimental Results

We experimented again on the human and the wheat dataset described in Section 3.4.1 and the experimental platform is also the Edison supercomputer. Our experiments sought to understand the strong scaling performance for these two datasets that differ by a factor of 6× in size.

5.5.1 Strong Scaling of End-to-End merAligner

Figure 5.7 shows the merAligner end-to-end strong scaling performance with all optimizations applied. This summarizes the main result of this chapter, and demonstrates the efficient utilization of distributed memory architectures for enabling scalable high performance sequence alignment. More specifically when scaling from 480 to 15,360 cores the total execution time drops from 4,147 seconds to 185 seconds (a 22× speedup), which translates to 0.7 parallel

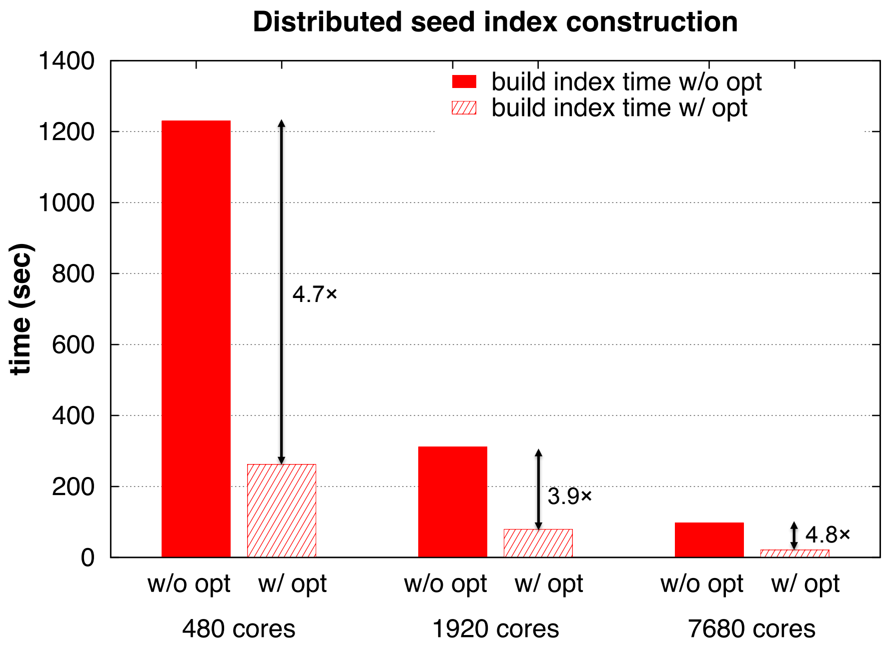


Figure 5.8: Distributed seed index construction scaling before and after applying the dynamic message aggregation optimization for the human dataset.

efficiency at the extreme scale for the human dataset (red curves). At the scale of 15,360 cores our approach performs alignment at 15,499,718 reads/sec. For the larger wheat data set (blue curves), scaling from 960 to 15,360 cores achieves 0.78 parallel efficiency. Note the super-linear speedup in the range of 960 — 7,680 cores, which we speculate is due to reduced congestion on the NIC since the communication is spread to even more nodes while we scale.

5.5.2 Anatomy of the Optimizations' Benefits

We now examine the individual effects of our optimization schemes, by selectively turning them off and measuring the resulting performance impact.

Distributed Seed Index Construction

Figure 5.8 illustrates the scaling of the distributed seed index construction before and after applying the "aggregating stores" optimization, given a size $S = 1000$ for the aggregating buffers. Observe that the reduction in the communication via our optimization dramatically decreases the construction time. At 480 cores the time spent decreases from 1,229 seconds to 262 seconds (4.7× improvement), and similarly at 7,680 cores we achieve an improvement of 4.8×. For the optimized construction phase, increasing concurrency from 480 cores to 7,680 (16× core increase) results in a near-linear speedup speedup of 12.7×. These results show that our algorithm efficiently parallelizes the seed index construction in a distributed

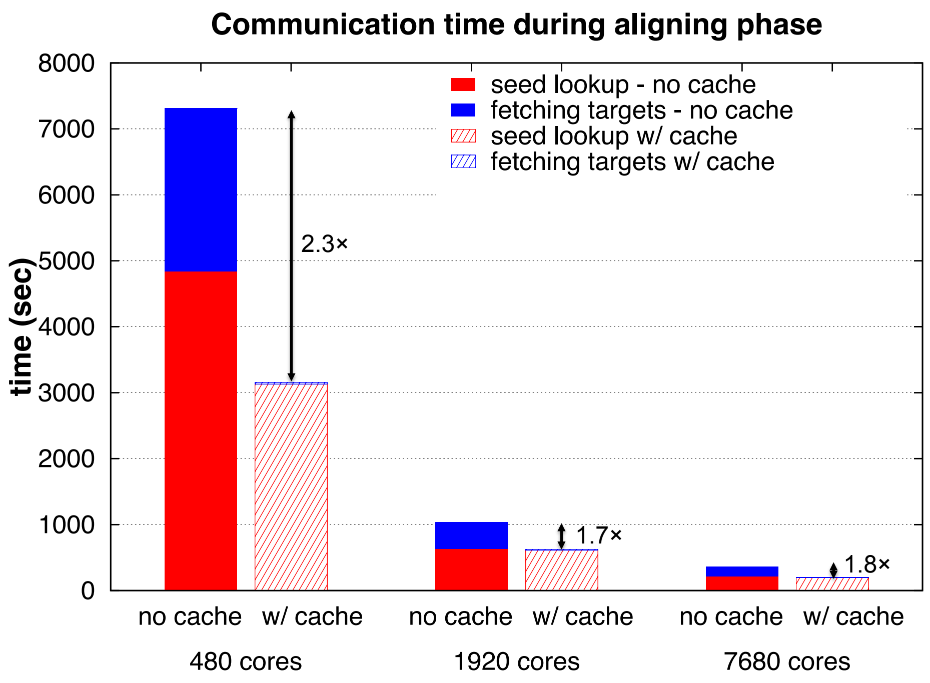


Figure 5.9: Impact of software caching on communication for the alignment of the human dataset.

memory and enables end-to-end scaling of merAligner. These scalable seed index construction results are in contrast to serial approaches of competing alignment codes as detailed in Subsection 5.5.3. Also, the algorithm achieves almost perfect load balance in terms of the number of distinct seeds assigned to each processor, thanks to our use of the `djb2` hash function to implement the seed to processor map.

Software Caching

In Figure 5.9 we depict the benefits of software caching on the communication time during the alignment phase (in all experiments 16 GB/node and 6 GB/node are allocated for the seed index and the target cache respectively). The red bars indicate the communication time for the seed lookups and the blue bars represent target sequence fetching overhead. Observe that the target cache is extremely efficient at all concurrencies and it essentially obviates all the communication involved with target sequences. Results also show that the seed index cache is effective at small concurrencies, where lookup time is decreased from 4,839 seconds to 3,130 seconds ($\approx$ 35% reduction) at 480 cores, whereas larger concurrencies see small benefits — validating our analysis in Subsection 5.2.2. Overall, the caching scheme decreases communication overhead by 2.3×, 1.7× and 1.8× at concurrencies of 480, 1,920 and 7,680 cores respectively.

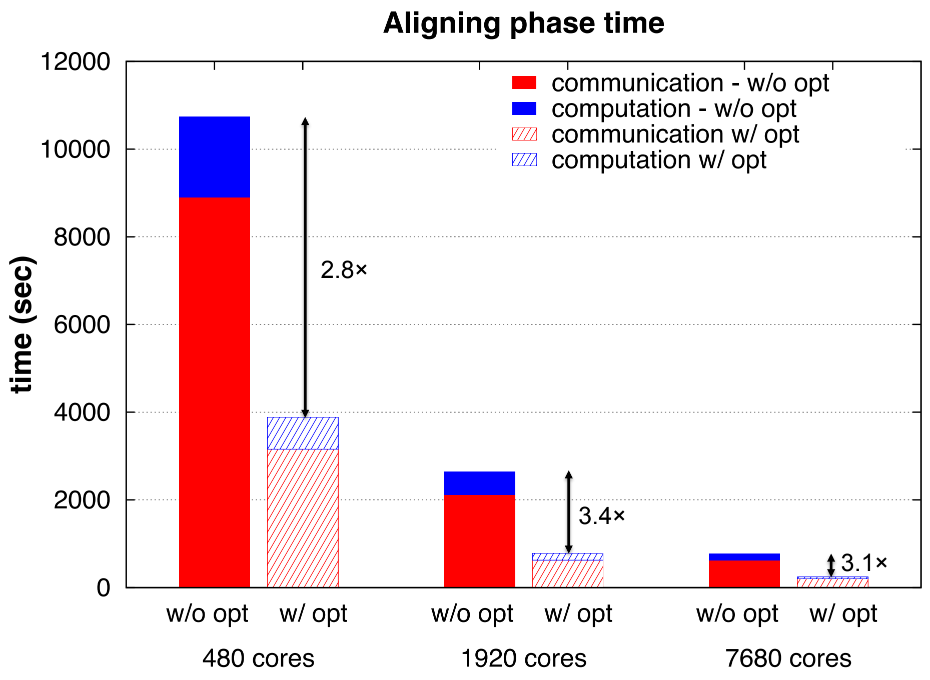


Figure 5.10: Impact of "exact matching optimization" on the aligning phase of the human dataset.

Exact Read Matching Optimization

Figure 5.10 shows the significant performance benefits of exact read matching, validating the theoretical analysis of Subsection 5.3.1. Here the optimization results in runtime improvement of the alignment step by factors of 2.8× and 3.1× for 480 cores and 7,680 cores respectively. Note that these gains come from both decreased communication (since in exact matching just one seed lookup is sufficient) and reduction of computation time (by avoiding Smith-Waterman execution). For example, at 480 cores our approach improves computation by 2.48× and communication by 2.82×. Finally we emphasize that ≈ 59% of the aligned reads took advantage of this optimization, thus enabling these impressive performance gains. For the optimized aligning phase, increasing concurrency from 480 cores to 7,680 (16× core increase) results in a near-linear speedup of 15.9×.

Load Balancing

In order to assess the effectiveness of the load balancing scheme, we conducted experiments with and without permuting the input read files and measured the maximum, minimum and average computation time as well as alignment times (computation plus communication). Results for 480 cores for the human dataset are shown in Table 5.1. Although our load balancing scheme effectively helps reduce the maximum computation time by almost 2.5×, the total alignment time is only improved by ≈ 5%. A closer investigation of the original

Load	Computation time			Total Alignment time		
Balancing	Min	Max	Avg	Min	Max	Avg
Yes	678	**800**	740	2700	**3885**	3277
No	515	**1945**	690	1512	**4092**	2073

Table 5.1: Effect of load balancing scheme on the human dataset, showing the reduction of the maximum compute time

dataset reveals that the reads mapping to the same genome region are grouped together. Since some groups of reads did not map to any target, they do not require Smith-Waterman execution, and thereby cause an imbalanced computing load. However, this locality made our seed index cache extremely effective and substantially decreased the communication time. Therefore, our load balancing scheme alleviates the computational load imbalance, while making the seed index cache less effective as seen in Table 5.1. Nonetheless, our approach improves the overall execution time. We note that the read grouping in the original data set is not the common case, and thus expect our load balancer to be even more effective in the general case.

5.5.3 Comparison with Existing Parallel Aligners

To assess our optimized merAligner in the context of existing solutions, we compare human data performance with BWA-mem [63] and Bowtie2 [60] using the pMap [88] framework. Note that pMap was modified to use the latest versions of the alignment software. Our experiments are configured using 4 instances of 6 threads per Edison node, since it is not possible to run one instance per core due to memory requirements of BWA-mem and Bowtie2 (each node contains 64GB of memory, which is insufficient to hold 24 instances of the seed index). BWA-mem is run with minimum seed length equal to 51 (like merAligner). For Bowtie2, we set the minimum seed length to the maximum possible value (31) and we execute the experiment with the `--very-fast` option in order to achieve the best mapping runtime. It is important to highlight that the seed index construction for BWA-mem and Bowtie2 is performed serially. For both cases, pMap partitions the reads to the available instances, then

Aligner	Seed Index Construction	Mapping Time	Total	Speedup
merAligner	21 *(P)*	263 *(P)*	**284 sec**	**1**×
BWA-mem	5,384 *(S)*	421 *(P)*	**5,805 sec**	**20.4**×
Bowtie2	10,916 *(S)*	283 *(P)*	**11,119 sec**	**39.4**×

Table 5.2: End-to-end performance comparison between parallel executions of merAligner, BWA-mem and Bowtie2 using 7,680 cores on the human data set (with all times in seconds) – highlighting serial *(S)* or parallel *(P)* implementation of the phases.

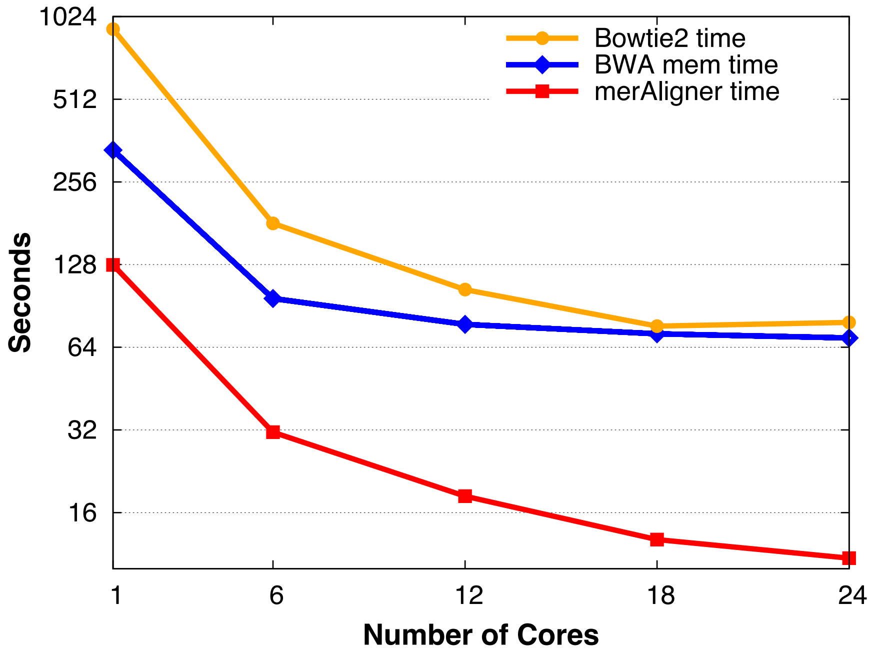


Figure 5.11: Shared memory performance of merAligner, BWA-mem and Bowtie2 on a single node of Edison on the *E. coli* data set. At the scale of 24 cores, merAligner is 6.33× faster than BWA mem and 7.2× faster than Bowtie2.

the seed index is loaded into each instance's memory and finally the corresponding instance is called on the set of reads assigned to it.

Table 5.2 presents comparative end-to-end performance results at 7,680 cores, and notes which computing phases are performed in serial *(S)* or parallel *(P)*. As expected, the serial seed index construction is a major bottleneck for the competing codes, compared with our parallel merAligner approach. Also, pMap spends a significant amount of time in read partitioning by having a single process sending the appropriate portion of the input read files to the corresponding node (4,305 and 3,982 seconds for BWA-mem and Bowtie2 respectively). On the contrary, merAligner does not suffer from this overhead since all processors read *in parallel* the appropriate portions of the input read files. To make though a fair comparison, we exclude the timing of the read partitioning for the cases of BWA-mem and Bowtie2. In total, merAligner is 20.4× and 39.4× faster than the parallel execution of BWA-mem and Bowtie2, respectively.

A complete analysis of the accuracy of the method is outside of the scope of this chapter. The algorithm is guaranteed to identify all alignments that share at least one identically matching stretch of at least $length(seed)$ consecutive bases between query and target sequences. Whether such alignments are sufficient is largely an application-dependent question. For the purposes of the Meraculous de novo assembly toolkit, these alignments are precisely those required. Here, we simply report all alignments detected (i.e. without any percentidentity thresholding, using a commonly employed scoring matrix) and find that merAligner successfully aligned 86.3% of the reads, while BWA-mem and Bowtie2 aligned 83.8% and 82.6% of the reads respectively. The accuracy of the detailed alignments is a function of the Smith-Waterman code, and we refer the interested reader to that publication [108].

To further assess merAligner performance, we conduct experiments on the smaller 4.64 Mbp *E. coli* K-12 MG1655 dataset, which allows single node scalability experiments using both BWA-mem and Bowtie2 in parallel mode with threads. The execution time of all three approaches (using a seed length of 19) is shown in Figure 5.11. Observe that merAligner performance continues to scale using all 24 available cores, while the runtimes of BWA-mem and Bowtie2 stop improving at 18 cores. Overall, merAligner is significantly faster, exceeding BWA-mem and Bowtie2 performance on 24 cores by 6.33× and 7.2× respectively. Subject to the alignment correctness discussion above, we find that merAligner successfully aligned 97.4% of the reads, while BWA-mem and Bowtie2 aligned 96.3% and 95.8% of the reads respectively.

5.6 Conclusion

In this Chapter we presented merAligner, a highly parallel sequence aligner that implements a seed–and–extend algorithm and employs parallelism in all of its components. MerAligner relies on a high performance distributed hash table (seed index) and uses one-sided communication capabilities of the Unified Parallel C to facilitate a fine-grained parallelism. We leverage communication optimizations at the construction of the distributed hash table and software caching schemes to reduce communication during the aligning phase. Additionally, merAligner preprocesses the target sequences to extract properties enabling exact sequence matching with minimal communication. Finally, we efficiently parallelize the I/O intensive phases and implement an effective load balancing scheme. Results show that merAligner exhibits efficient scaling up to thousands of cores on a Cray XC30 supercomputer using real human and wheat genome data while significantly outperforming existing parallel alignment tools.

Chapter 6

Parallel Scaffolding and Gap Closing

The main goal of the scaffolding algorithm in our pipeline is to connect together contigs and form *scaffolds* which are long sequences of contigs. The first step of scaffolding comprises of assessing the contigs created by contig generation and using the branches of the full de Bruijn graph (i.e. the “fork” k-mers) to discover information regarding the connectivity among contigs. By exploiting this connectivity information we generate a graph of contigs which is further processed in order to be simplified, e.g. by identifying and transforming special graph structures. Then, by leveraging the reads-to-contigs alignments and the information from paired reads we introduce additional links/edges in the contig graph. Note that paired reads with large insert sizes can be used to generate long-range links among contigs that could not be found from the k-mer de Bruijn graph. Afterwards, we traverse the updated contig graph and form chains of contigs that constitute the final scaffolds. It is possible though that there are gaps between pairs of contigs. Therefore, we further process the reads-to-contigs alignments and locate the reads that are placed into these gaps. Ultimately, we leverage these subsets of reads and close the gaps by performing a mini-assembly algorithm involving only the localized reads for each gap. The outcome of this step constitutes the result of the Meraculous assembly pipeline.

In this chapter we describe in detail the parallelization of the scaffolding and the gap closing algorithms. This material was first published in the paper “HipMer: An Extreme-Scale De Novo Genome Assembler” [33]. The gap closing parallelization is done by Steven Hofmeyr who co-authored that paper.

6.1 Computing Contig Depths

First, we calculate the depth (coverage) of each contig C_i as the average of the k-mer counts in C_i. This information is important because it dictates contigs with low coverage and contigs that represent repetitive genome regions; the latter type of contigs typically have much larger coverage than the rest contigs. The contig depth information is used in subsequent steps of the scaffolding algorithm.

Given the exact count (depth) of the k-mers and a set of contigs, we calculate the depth of each contig C_i as:

$$depth(C_i) = \frac{\displaystyle\sum_{k\text{-}mer_j \in C_i} count(k\text{-}mer_j)}{length(C_i) - k + 1} \tag{6.1}$$

This information is important in order to approximate the read coverage of each contig and will be used in later scaffolding stages. The parallelization here consists of two steps:

1. The k-mers are stored in a distributed hash table where the keys are k-mers and the values are the corresponding counts. We recognize this computational pattern as the Use Case 1 (GUO) of the distributed hash tables (see subsection 2.3.2), therefore we can mitigate the communication and synchronization overheads by leveraging dynamic message aggregation (see subsection 4.1.1 for more details).

2. Each processor p_i is assigned $1/p$ of the contigs (p is the total number of available processors) and for every contig, p_i looks up all the contained k-mers and sums up their counts. The depth of the contig is then calculated as the mean count of all the k-mers (see equation 6.1). Note that this step does not require any synchronization, as the distributed hash table is only read after its construction.

6.2 Identifying Termination States of Contigs

In this step we identify the termination condition for every contig generated by the de Bruijn graph traversal. A contig may have been terminated because: (1) it does not have any neighboring `UU` k-mers to its endpoints, or (2) one of its endpoints may be adjacent to a "fork" k-mer (see Figure 2.3 for an example of such a "fork" k-mer). The termination states of contigs provide information regarding the connectivity among contigs and will be used in subsequent scaffolding modules that operate on graphs of contigs.

In the distributed hash table of k-mers we store their corresponding extensions as their values. Again we identify the Use Case 1 (GUO) of the distributed hash tables (see subsection 2.3.2) and we aggregate dynamically the messages. The algorithm then iterates in parallel over the contig set and identifies the termination condition for every contig. For example, a contig may have been terminated because it did not find any neighboring `UU` k-mers to its endpoints, or it may have found a branch in the graph with two high quality neighboring k-mers. The latter is the case that often arises in: (i) diploid organisms, i.e. genomes that have two sets of chromosomes, one set inherited from each "parent" and (ii) in repetitive genome regions. The Meraculous assembler terminates contigs if they do not have unique high quality neighboring k-mers, but this termination state is utilized in the following bubble identification step.

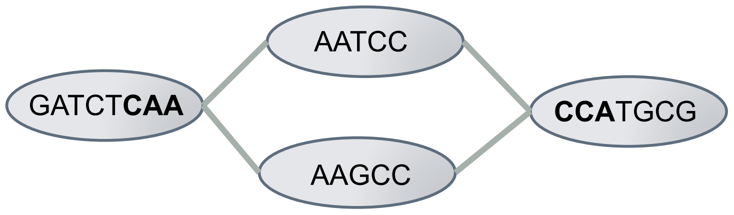


Figure 6.1: Example of a bubble structure in a bubble-contig graph.

6.3 Identifying Bubble Structures

In this step, we examine the termination condition of contigs from the previous step and we detect *bubbles*: contigs that have the same k-mers as extensions of their endpoints. Note that this is only applicable in diploid genomes. By leveraging this bubble information and the contigs's end termination state, a graph of contigs is created, called the *bubble-contig* graph. Figure 6.1 illustrates an example of such a bubble structure. Assume that all four contigs `GATCTCAA`, `AATCC`, `AAGCC`, `CCATGCG` have been generated by a de Bruijn graph of k-mers with $k = 3$. Also, observe that the k-mers `CAA` and `CCA` (which are highlighted with boldface text) are "fork" k-mers and they can be used to recognize this bubble structure. This graph is orders of magnitude smaller than the original k-mer de Bruijn graph because the connected components (contigs) of the de Bruijn graph have been contracted to supervertices. We build this bubble-contig graph in parallel by employing a distributed hash table.

Once the bubble-contig graph is built, it is traversed to merge qualifying contigs (e.g. by following one of the paths in the bubbles). In the example of Figure 6.1 by following the upper path we get a the single contig `GATCTCAATCCATGCG`. We parallelize this bubble-contig graph traversal using a speculative approach. The processors pick random seeds (in this case the traversal seeds are contigs) from the bubble-contig graph and initiate independent traversal. Once an independent traversal is terminated we store the resulting path as a new contig. However, if multiple processors work on the same path, they abort their traversals and allow a single processor to complete them. In practice, this speculative execution spends most of the time ($\sim 99\%$) in parallel traversals. Given that the bubble-contig graph is orders of magnitude smaller than the original de Bruijn graph, its traversal is extremely fast. The result of this module is a set of contig paths, where every path can be compressed to a single DNA sequence. For simplicity, we call these compressed paths also contigs in the description of the following scaffolding modules.

6.4 Alignment of Reads onto Contigs

In this pipeline phase the goal is to map the original reads onto the generated contigs; the contigs can be generated either by the contig generation step or by the bubble-contig graph simplification in the case of diploid genomes. This mapping provides information about the

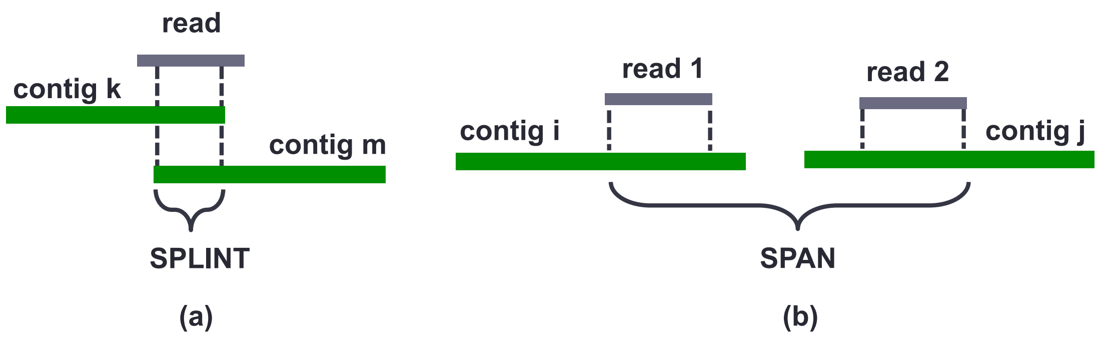


Figure 6.2: (a) Contigs k and m constitute a splint. (b) Contigs i and j constitute a span.

relative ordering and orientation of the contigs. In order to efficiently parallelize the reads-to-contigs alignment, we employ merAligner [34] described in more detailed in Chapter 5, a scalable, end-to-end parallel sequence aligner. The seed length used in merAligner has the same value as the k used for k-mers in the de Bruijn graph.

6.5 Insert Size Estimation of Read Libraries

Given a set of paired reads from a library (i.e. the reads come in pairs) and the corresponding read-to-contig alignments from the previous step, we now use full length alignments in which both reads of a pair are placed within a common contig, and calculate the insert size. Even if the library is characterized by an insert size (see Figure 2.1) we want to get a more accurate estimate for the mean of its value and its standard deviation in order to be more precise in the scaffolding decisions where the insert size plays a crucial role. Sampling the alignments allows us to estimate the insert size of the library in an efficient way. The insert size estimation is parallelized by having p processors build local histograms of distinct sampled alignments and eventually merging these p local histograms to a global one, from which the mean insert size and the standard deviation of the read library are computed.

6.6 Locating Splints and Spans

The next step is to process the alignments and identify *splints*, which are contigs that overlap at their ends. Essentially, if a particular segment of a read aligns to the ends of two different contigs we conclude that these contigs form a splint (see Figure 6.2 (a)). The parallelization of this step is straightforward: Each one of the p processors independently processes $1/p$ of the total read alignments and stores the splints's information it identifies. The computation of this step is embarrassingly parallel.

Additionally, by processing paired reads' alignments we identify *spans*, which are pairs of contigs associated with particular read pairs via their read-to-contig alignments. For example, consider that the first read of a pair aligns with contig i while the second read of that pair aligns with contig j. It can thus be concluded that contigs i and j form a span (see Figure 6.2 (b)). Note that we have already calculated the insert size of the read library in the previous step and therefore can estimate the gap size between contigs i and j. Again the parallelization of this module is straightforward, where each processor independently assesses $1/p$ of the total read alignments and stores the spans's information.

6.7 Contig Link Generation

Once splints and spans are created, they can be processed to generate *links* among pairs of contigs. More specifically, if a sufficient number of read alignments supports a particular splint between contig k and contig m, we generate a *splint link* for that pair of contigs. Parallelizing this operation requires a distributed hash table, where the keys are pairs of contigs and values are the splint/overlap information. Each processor is assigned $1/p$ of the splints and stores them in the distributed hash table. Here, we again apply the aggregating stores optimization to minimize the number of messages and the synchronization cost (Use Case 1 (GUO) 2.3.2 of distributed hash tables). When all splints have been stored in the distributed hash table, each processor iterates over its *local buckets* to further assess/count the splint links (Use Case 4 (LRW) 2.3.2 of distributed hash tables).

In an analogous way, if a sufficient number of paired reads' alignments supports a particular span between contig i and contig j we generate a *span link* for that pair of contigs. The parallelization of this operation relies on a distributed hash table where the keys are pairs of contigs and values are the span/gap information. Again we have each processor process $1/p$ of the spans and apply the aggregating stores optimization. After the distributed hash table construction is finalized, each processor iterates over its local entries to further assess/count the span links.

6.8 Ordering and Orientation of Contigs

After storing the previously generated links in a distributed hash table (Use Case 1 of distributed hash tables with dynamic message aggregation) where the keys are pairs of contigs and values are the corresponding link information, the data in the links is processed to build *ties* among the contigs by consolidating splint/span links. If the contigs that are about to be tied differ significantly in their depth, we can reject that tie. Also, the depth information can be used to recognize contigs arising from repetitive regions: such contigs should have significantly larger depth than the depth of coverage d. Typically such contigs have multiple links on both of their endpoints.

This step employs parallelism by having each processor work on the local buckets of the links' distributed hash table (Use Case 4 of distributed hash tables). After the establishment of ties among the contigs, we traverse the implicit graph of ties and then lock contigs

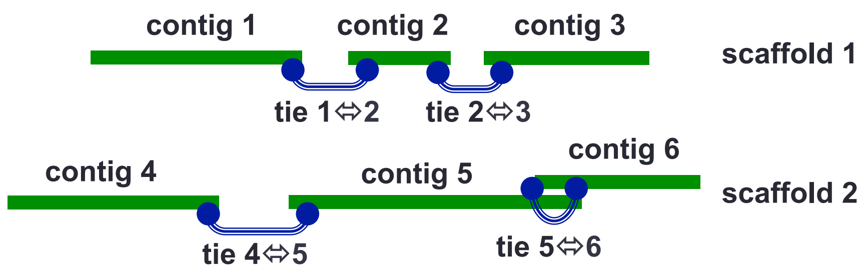


Figure 6.3: Two scaffolds formed by traversing sequences of ties. Note that the tie between contigs 5 and 6 is due to a splint-link while the rest ties are due to span-links.

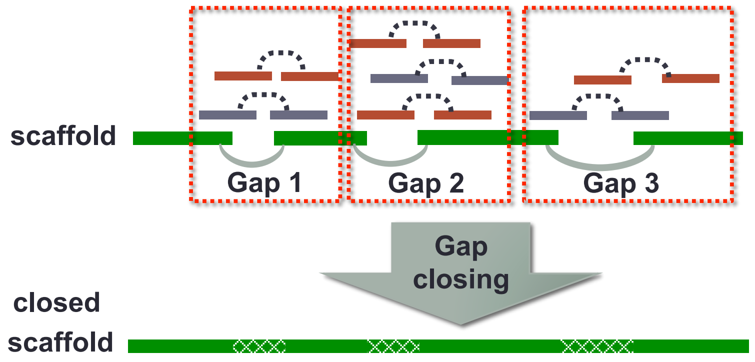


Figure 6.4: A set of gaps in a scaffold to be closed.

together in order to form *scaffolds* (see Figure 6.3). The traversal is done by selecting seeds (traversal seeds are contigs in this case) in order of decreasing length and therefore it is inherently serial; this heuristic tries to lock together first “long” contigs. We have optimized this component and found that its execution time is insignificant compared to the previous pipeline operations. This behavior is expected since the number of contigs (vertices in the ties’s graph) is orders of magnitude smaller than the number of k-mers (vertices in the de Bruijn graph of the pipeline).

Finally, libraries with large insert sizes can be used to generate long-range links/ties and consequently to resolve repeats in the contig graph. To accommodate this functionality, the scaffolding algorithm can be performed in an iterative way by using links generated from different libraries at each iteration.

6.9 Gap Closing

The gap closing stage uses the merAligner outputs (i.e. read-to-contig alignments), the scaffolds and the contigs to attempt to assemble reads across gaps between the contigs of scaffolds (see Figure 6.4). To determine which reads map to which gaps, the alignments are processed in parallel and projected into the gaps. The gaps are divided into subsets and each set is processed by a separate thread, in an embarrassingly parallel phase. Several methods are available for constructing closures, and are used in succession until a closure is found or no more methods are available. The first method used is *spanning*, i.e. finding a read that begins with the end of the contig on one side of the gap, and finishes with the start of the contig on the other. Should spanning fail, an attempt is made to do a traversal (k-mer walk) across the reads from one side of the gap to the other, with iteratively increasing k-mer sizes until the gap is closed. This mini-assembly step is first attempted from the right hand side contig to the left, and if that fails, from left to right. If both traversals fail, the final method used is an attempt to *patch* across the two incomplete traversals, i.e. find an acceptable overlap between the two sequences.

The various closure methods differ in computational intensity, with spanning and patching being orders of magnitude quicker than k-mer walks. Given that it is not clear *a priori* what methods will be successful for closing a gap, the computational time can vary by orders of magnitude from one closure to the next. To prevent load imbalance in the gap closing phase, the gaps are distributed in a Round Robin fashion across all the available threads. This suffices to prevent most imbalance because it breaks up the gaps from a single scaffold, which tend to require similar costs to close.

6.10 Experimental Results

The experiments presented in this section utilize the human and wheat datasets on the Edison supercomputer.

Figure 6.5 illustrates the strong scaling performance for the scaffolding module using the human dataset. This graph depicts three scaffolding component runtimes at each concurrency: (1) the merAligner runtime, the most time consuming module of scaffolding, (2) the time required by the gap closing module and (3) the execution time for the remaining steps of scaffolding. The final component includes computation of contig depths and termination states, identification of bubbles, insert size estimation, location of splints & spans, contig link generation and ordering/orientation of contigs — which have all been parallelized and optimized. Scaling to 7,680 cores and 15,360 cores results in parallel efficiencies (relative to the 480 core baseline) of 0.48 and 0.33, respectively. While merAligner exhibits effective scaling behavior all the way up to 15,360 cores (0.64 efficiency), the scaling of gap closing is more constrained by I/O and only achieves a parallel efficiency of 0.35 at 7,680 cores, and 0.19 at 15,360 cores. Similarly, the remaining modules of scaffolding show scaling up to 7,680 cores, while once again suffering from I/O saturation at the highest (15,360 cores) concurrency.

Figure 6.6 presents the strong scaling behavior of scaffolding for the larger wheat dataset. We attain parallel efficiencies of 0.61 and 0.37 for 7,680 and 15,360 cores respectively (relative to the 960 core baseline). While merAligner and gap closing exhibit similar scaling to the human test case, the remaining scaffolding steps consume a significantly higher fraction

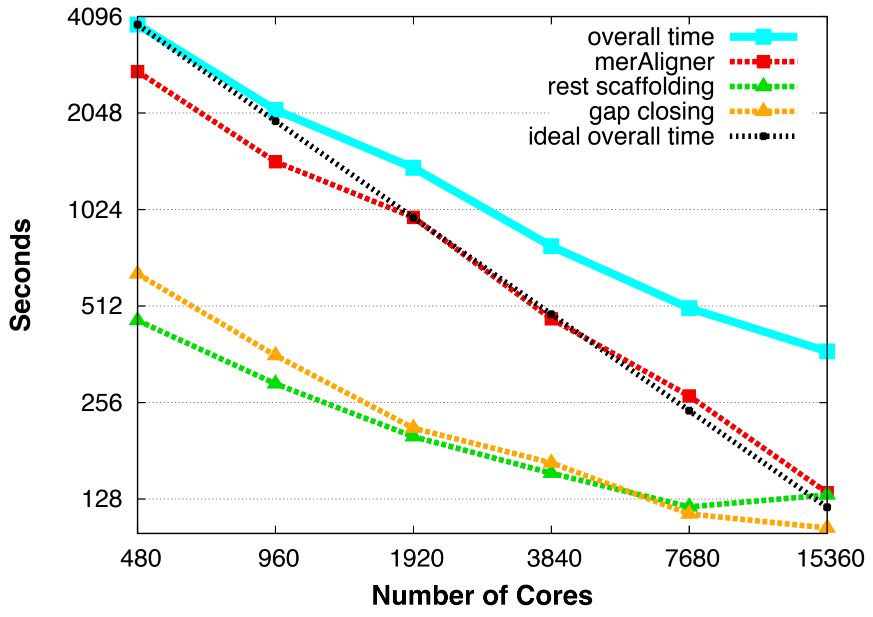


Figure 6.5: Strong scaling of scaffolding for the human genome. Both axes are in log scale.

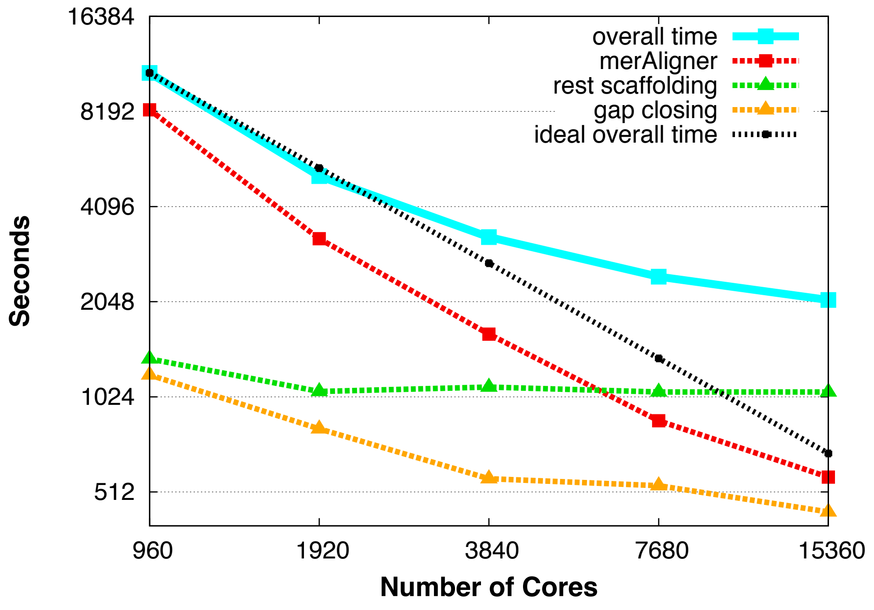


Figure 6.6: Strong scaling of scaffolding for the wheat genome. Both axes are in log scale.

of the overall runtime. There are two main reasons for this behavior. First, the highly repetitive nature of the wheat genome leads to increased fragmentation of the contig generation compared with the human DNA, resulting in contig graphs that are contracted by a smaller fraction. We refer to contraction with respect to the size reduction of the k-mer de Bruijn graph when it is simplified to equivalent contig graph. Hence, the serial component of the contig ordering/orientation module requires a relatively higher overhead compared with the human dataset. Second, the execution of the wheat pipeline as performed in our previous work [20] requires four rounds of scaffolding with libraries of different insert sizes, resulting in even more overhead within the contig ordering/orientation module.

6.11 Conclusion

In this chapter we described in detail the parallel algorithms that constitute the scaffolding and gap closing modules in the HipMer pipeline. In the parallelization process we identified properties of the underlying distributed data structures (hash tables) that enable communication optimizations. We also developed a speculative algorithm for the bubble identification step and we parallelized efficiently the time consuming gap closing procedure. Most of the time in scaffolding is required for sequence alignment and we employed for this purpose the merAligner algorithm. Finally, empirical large-scale results demonstrate that the parallel scaffolding module and gap closing can scale to 15K cores of the Edison supercomputer.

Chapter 7

Single Genome Assembly with HipMer

The primary goal of this dissertation is to provide a high-quality, high-performance *de novo* single genome assembler. Therefore we focused on the parallelization of a state-of-the-art assembly pipeline called Meraculous. The parallelization of the end-to-end pipeline has been described in detail in Chapters 3, 4, 5 and 6 and the outcome is a fully parallel assembler we call HipMer. In this Chapter[1] we discuss first the quality of HipMer and illustrate that it provides high-quality assemblies in a set of benchmarks. Then, we provide a rough performance model for the whole pipeline that gives insights for the relative costs of the various components. Finally, we present performance results for the end-to-end pipeline and compare its execution time and scalability with other parallel de novo assembly pipelines.

7.1 Quality Assessment of HipMer Assemblies

The HipMer pipeline is a faithful implementation of the original Meraculous algorithm. Therefore, in order to assess the quality of the assemblies in comparison to other state-of-the-art tools we refer to the most recent Assemblathon study [13] where Meraculous is a participating team; detailed quality analysis of the Meraculous assembler is out of scope of this dissertation.

Table 7.1 shows the participating teams along with the assembly software they used. Each team was allowed to submit one competitive entry for each of the three datasets (bird, fish, and snake) that were used as the quality benchmark in this study. Figure 7.1 shows the quality results in regard to the REAPR [49] summary score; this metric rewards short-range and long-range accuracy, as well as low error rates. This score is calculated as the product of i) the number of error free bases and ii) the squared scaffold N50[2] length after breaking assemblies at scaffolding errors divided by the original scaffold N50 length. This metric takes

[1]The material presented in this Chapter was first published in the paper "HipMer: An Extreme-Scale De Novo Genome Assembler" [33].

[2]N50 is calculated by summing all sequence lengths, starting with the longest, and observing the length that takes the sum length past 50% of the total assembly length

Team-ID	Assembly software used	Team-ID	Assembly software used
ABL	HyDA	CURT	SOAPdenovo, velvet
ABYSS	ABySS and Anchor	GAM	GAM, CLC and ABySS
ALLP	ALLPATHS-LG	IOB	ALLPATHS-LG and SOAPdenovo
BCM	Quake, ALLPATHS-LG, Velvet	MLK	ABySS
CBCB	Celera assembler	MERAC	Meraculous
COBIG	Mira, Bambus2	NEWB	Newbler
CRACS	ABySS, SSPACE, Bowtie, and FASTX	PHUS	Phusion2, SOAPdenovo, SSPACE
CSHL	ALLPATHS, SOAPdenovo	PRICE	PRICE
CTD	Unspecified	RAY	Ray
SGA	SGA	SOAP	SOAPdenovo
SYMB	SSPACE, SuperScaffolder & GapCloser		

Table 7.1: Participating teams in the Assemblathon 2 study. For more details on the employed software we refer the interested reader to the Assemblathon 2 study [13].

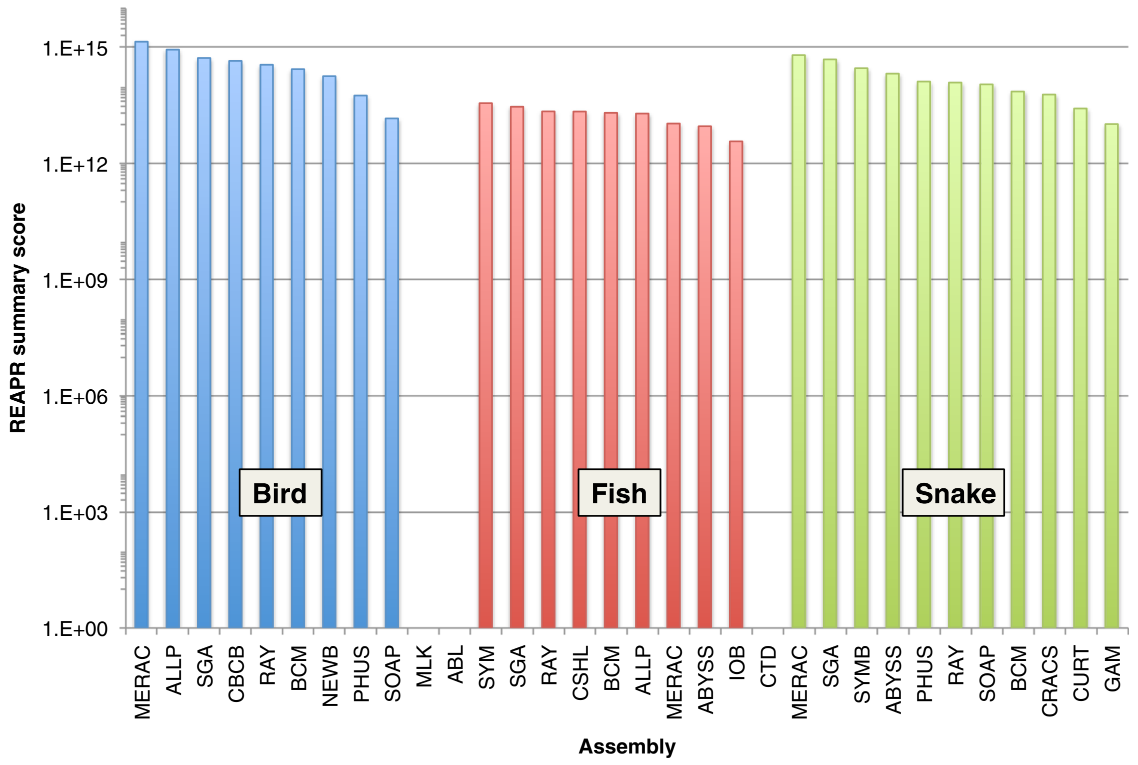


Figure 7.1: REAPR summary scores for all assemblies. This score is calculated as: `number of error free bases` $\times$ `(broken N50)`^{2}`/original N50`. Data shown for assemblies of bird (blue), fish (red), and snake (green). The REAPR summary score is plotted on a log axis. Results for bird assemblies MLK and ABL and fish assembly CTD are not shown as it was not possible to run REAPR on these assemblies. Assemblies were excluded from the analysis if their total size was less than 25% of the expected genome size for the species in question. This plot is taken from the Assemblathon 2 study [13].

into account both the contiguity and the accuracy of the assemblies and therefore we opt to present it here as representative of the overall results.

The REAPR summary scores reveal large differences between the quality of different assemblies. The Meraculous assembler is ranked first for the bird and the snake assemblies while it is ranked seventh in regard to the fish assembly, where in general all assemblers perform worse compared to the other two datasets. It is worth noting that the REAPR score considers both accuracy and contiguity. For example, the snake Meraculous and SGA assemblies have comparable numbers of error free bases, but the N50 before and after breaking at errors makes the difference between their final scores. Although the Meraculous assembly had lower N50 values than those of the SGA assembly, it made proportionally fewer errors, so that it was ranked above SGA. Of course, the REAPR quality score is not a panacea, it just highlights in this case study that the Meraculous assembler (and consequently HipMer) performs well compared to other state-of-the-art tools. We refer the interested reader to the Assemblathon 2 paper for a comprehensive analysis of the quality results.

7.2 Performance Modeling of the HipMer Pipeline

This section provides a rough performance model for the whole pipeline that gives insights for the relative costs of the various components. In particular, we consider only the computational costs and we do not take into account parallelization overheads such as communication, synchronization and load imbalance. Nevertheless, we have shown in the previous Chapters that all these modules can be parallelized efficiently and we can minimize synchronization, communication and load imbalance. The HipMer pipeline is illustrated in Figure 7.2 and consists of four modules: (1) k-mer analysis, (2) contig generation, (3) sequence alignment and (4) scaffolding & gap closing. In the following subsections we provide *lower bounds* of the computational cost of each one of these steps which are necessarily lower bounds on the total cost, including communication, synchronization and load imbalance overheads. In other words, these costs are indicative of the actual complexity of the pipeline given work-efficient algorithms. Throughout the analysis we assume that the underlying genome has size G and the average coverage depth is d. We also assume that the read length is L and therefore the number of reads covering the genome is roughly $G \cdot d/L$.

7.2.1 Complexity of k-mer analysis

Given a parameter k for the k-mer length, from each read we extract $(L-k+1)$ k-mers. We have to access all these k-mers in order to count/characterize them, therefore the three-pass k-mer analysis step as described in Chapter 3 has roughly cost:

$$\mathrm{T}_{k\text{-mer analysis}} = \Omega(\#\text{ passes } \cdot \#\text{ reads} \cdot k\text{-mers per read}) = \Omega(3 \cdot G \cdot d/L \cdot (L-k+1)) \quad (7.1)$$

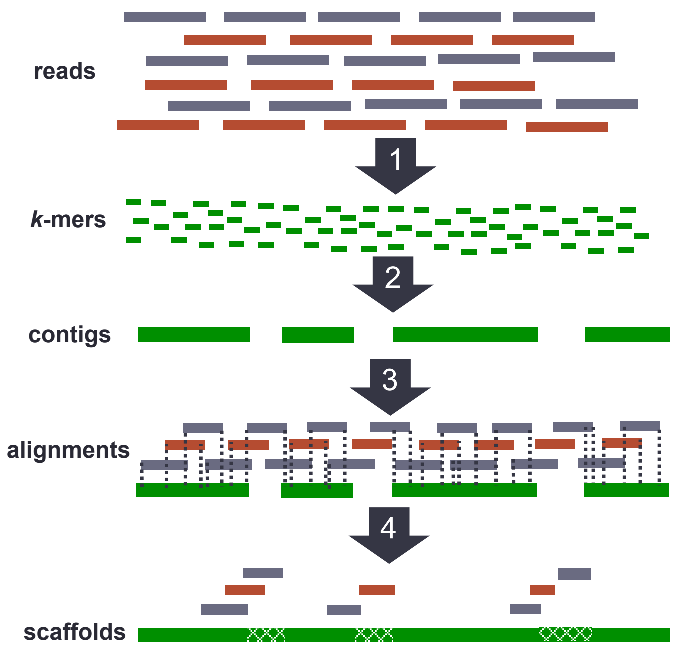


Figure 7.2: The HipMer *de novo* genome assembly pipeline.

7.2.2 Complexity of contig generation

Once the k-mer analysis has been performed, the multiple occurrences of k-mers have been consolidated to unique copies and they are used to form a de Bruijn graph which has size proportional to the genome size G. Also, all the erroneous k-mers have been filtered our during the previous step. Therefore, traversing the de Bruijn graph and discovering the connected components has cost:

$$\mathrm{T}_{\text{contig generation}} = \Omega(\text{size of de Bruijn graph}) = \Omega(G) \tag{7.2}$$

7.2.3 Complexity of sequence alignment

In this step we map the original read pairs onto the generated contigs. The first part of the alignment step is to generate an index on the contigs. The cost for constructing the index is proportional to the size of the contigs which have size roughly equal to the genome size; therefore the indexing cost is:

$$\mathrm{T}_{\text{contig indexing}} = \Theta(\text{genome size}) = \Theta(G) \tag{7.3}$$

Once we build the index, we lookup seeds which are extracted from the reads and we perform local alignments via banded Smith-Waterman. For our analysis we assume a band of width β in the Smith-Waterman execution. Also, for simplicity we assume that each read aligns on average onto α contigs and we can efficiently identify contig properties to avoid redundant computation and seed lookups as we described in Chapter 5. Therefore, the mapping step has cost:

$$\begin{aligned} \mathrm{T}_{\text{mapping}} &= \Omega(\#\text{ of reads} \cdot \#\text{ alignments per read} \cdot \text{cost per local alignment}) \\ &= \Omega(G \cdot d/L \cdot \alpha \cdot \beta \cdot L) = \Omega(G \cdot d \cdot \alpha \cdot \beta) \end{aligned} \tag{7.4}$$

The total cost for the sequence alignment module is therefore:

$$\mathrm{T}_{\text{sequence alignment}} = \Omega(\mathrm{T}_{\text{contig indexing}} + \mathrm{T}_{\text{mapping}}) = \Omega(G(1 + d \cdot \alpha \cdot \beta)) \tag{7.5}$$

7.2.4 Complexity of scaffolding

In order to simplify our cost analysis, we refer to the modules "Computing Contig Depths", "Identifying Termination States of Contigs" and "Identifying Bubble Structures" as *contig graph assessment*. These modules process the contig graph and assess the depth of coverage, the connectivity and the "bubble" structures respectively. Therefore the cost is:

$$\mathrm{T}_{\text{contig graph assessment}} = \Theta(3 \cdot \text{contig graph size}) = \Theta(3G) \tag{7.6}$$

In the remaining scaffolding modules, we analyze the computed alignments in order to generate links among contigs. Afterwards, we traverse the underlying graph of links/ties to generate scaffolds. The cost is therefore:

$$\begin{aligned} \mathrm{T}_{\text{links assessment}} &= \Omega(\mathrm{T}_{\text{alignments analysis}} + \mathrm{T}_{\text{traverse link graph}}) \\ &= \Omega(\#\text{ of alignments} + \text{size of link graph}) = \Omega(G \cdot d/L \cdot \alpha + G) \end{aligned} \tag{7.7}$$

Eventually we get the cost for the complete scaffolding step:

$$\mathrm{T}_{\text{scaffolding}} = \Omega(T_{\text{contig graph assessment}} + T_{\text{links assessment}}) = \Omega(G \cdot d/L \cdot \alpha + 4G) \tag{7.8}$$

7.2.5 Complexity of Gap Closing

With regard to the final gap closing module we assume that a fraction γ of the genomic sequences is not assembled into contigs and therefore the corresponding reads will be assembled via the gap closing process. Also, we assume that the average coverage in these regions is also d. Therefore, by performing a mini assembly algorithm on these reads and considering the most time consuming parts in this process, namely k-mer counting and k-mer walks, the computational cost is:

$$\begin{aligned} \mathrm{T}_{\text{gap closing}} &= \Omega(\mathrm{T}_{k\text{-mer counting}} + \mathrm{T}_{k\text{-mer walks}}) \\ &= \Omega(\#k\text{-mers in unassembled genome} + \text{size of unassembled genome}) \\ &= \Omega(\gamma G \cdot d/L \cdot (L - k + 1) + \gamma G) = \Omega(\ \gamma G(1 + d/L \cdot (L - k + 1))\ ) \end{aligned} \tag{7.9}$$

Symbol	Parameter Description
G	genome size
d	depth of coverage
L	read length
k	length of k-mers
α	average number of alignments per read
β	width of band in Smith-Waterman
γ	% of genome not in assembled contigs

Table 7.2: Model parameters

Stage	Complexity
k-mer analysis	$3 \cdot G \cdot d/L \cdot (L - k + 1)$
contig generation	G
sequence alignment	$G \cdot (1 + d \cdot \alpha \cdot \beta)$
scaffolding	$G \cdot d/L \cdot \alpha + 4G$
gap closing	$\gamma G \cdot (1 + d/L \cdot (L - k + 1))$

Table 7.3: Complexity of the pipeline

7.2.6 Complexity of the end-to-end pipeline

The results of the complexity analysis (equations 7.1 - 7.9) are summarized in Tables 7.2 and 7.3. In particular, Table 7.2 lists the parameters used in our complexity model and Table 7.3 illustrates the complexity of the main stages in the pipeline. Now we pick a few representative values for the parameters and overview the cost of the various stages of the pipeline. By setting $d = 50$, $k = L/2 + 1$, $\alpha = 1$, $\beta = 5$, $L = 150$ and $\gamma = 0.1$ we get:

$$\begin{aligned}
\mathrm{T}_{k\text{-mer analysis}} &= 75 \cdot G \\
\mathrm{T}_{\text{contig generation}} &= G \\
\mathrm{T}_{\text{sequence alignment}} &= 251 \cdot G \\
\mathrm{T}_{\text{scaffolding}} &= 4.3 \cdot G \\
\mathrm{T}_{\text{gap closing}} &= 2.6 \cdot G
\end{aligned}$$

We conclude that the most time consuming phases are k-mer analysis and sequence alignment, which both have orders of magnitude larger cost compared to the remaining steps in the pipeline. Of course this analysis is not rigorous but it gives an approximate estimate with regard to the relative costs of the HipMer components. One of the limitations is that the model ignores the parallelization overheads of the various computational patterns. For example, in Chapters 3 and 4 we saw that the k-mer analysis step follows a bulk synchronous parallel model while the contig generation has an inherently irregular pattern and as a result its parallelization is more challenging. Furthermore, this model is agnostic to the relative cost of the various operations (e.g. string computations vs I/O). For instance, the k-mer analysis has to deal with the I/O of the massive read datasets with size $\Theta(G \cdot d)$, while the contig generation takes as input a filtered k-mer set with size $\Theta(G)$. This gap in the expensive I/O phase will contribute more in the actual difference of the two stages' cost compared to differences arising from in-memory computations.

7.3 Performance Results of the end-to-end Pipeline

Figures 7.3 and 7.4 show the end-to-end strong scaling performance of HipMer (including I/O) with the human and the wheat datasets respectively on the Edison supercomputer. For

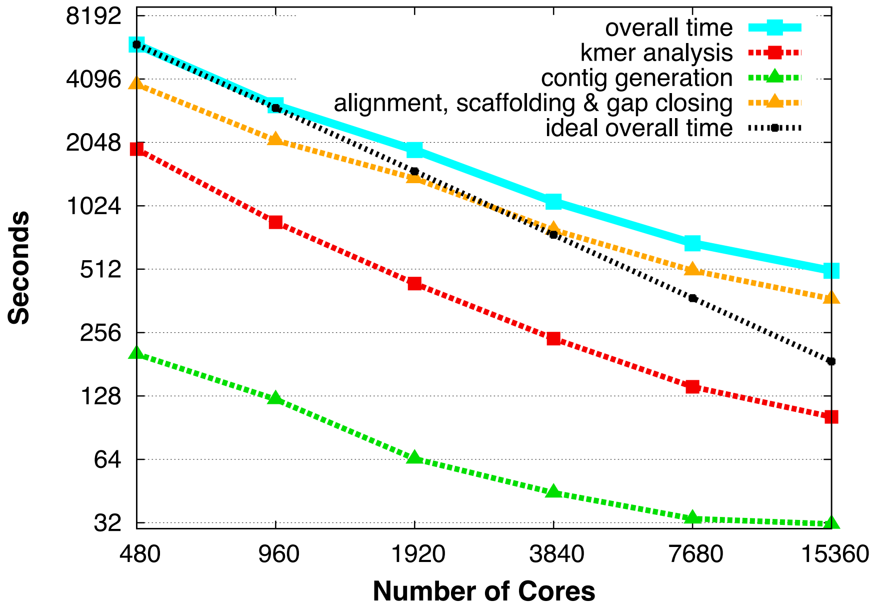


Figure 7.3: End-to-end strong scaling for the human genome. Both axes are in log scale.

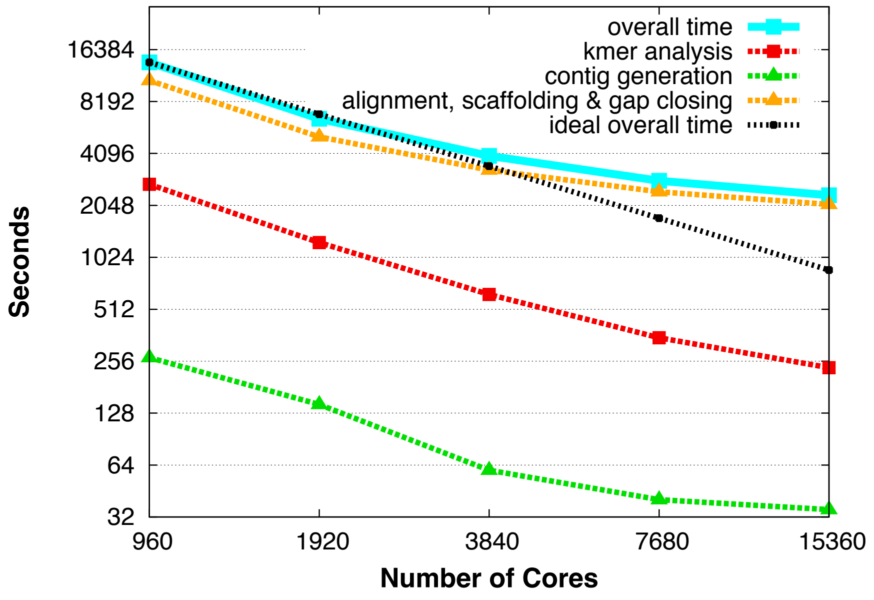


Figure 7.4: End-to-end strong scaling for the wheat genome. Both axes are in log scale.

the human dataset at 15,360 cores we achieve a speedup of 11.9× over our baseline execution (480 cores). At this extreme scale the human genome can be assembled from raw reads in just ≈ 8.4 minutes. On the complex wheat dataset, we achieve a speedup up to 5.9× over the baseline of 960 core execution, allowing us to perform the end-to-end assembly in 39

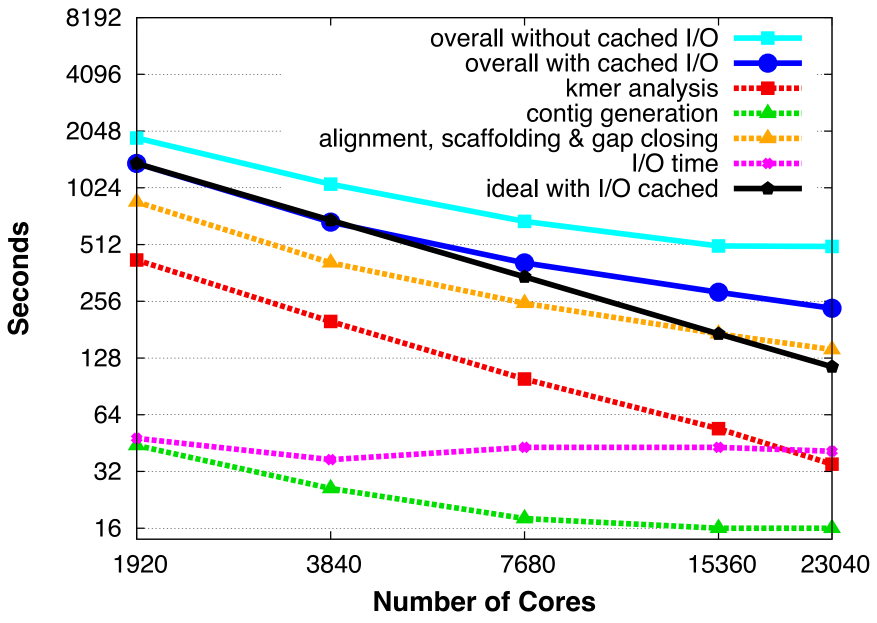


Figure 7.5: Strong scaling of the human data set with I/O caching. Both axes are in logarithmic scale.

minutes when using 15,360 cores. In the end-to-end experiments, a significant fraction of the execution time is spent in parallel sequence alignment, scaffolding and gap closing (e.g. 68% for human at 960 cores); k-mer analysis requires less runtime (28% at 960 cores) and contig generation is the least expensive computational component (4% at 960 cores). These empirical results support the model we presented in the previous section.

7.3.1 I/O caching

Our modular design of the pipeline enables flexible configurations that can be adapted appropriately to meet the requirements of each assembly. For instance, one might want to perform multiple rounds of scaffolding to facilitate the assembly of highly repetitive regions or to iterate over the k-mer analysis step and contig generation multiple times (with varying k and other parameters) in order to extract information that is latent within a single iteration. These configurations imply that the input read datasets should be loaded multiple times. Even in a typical, single pass execution of the pipeline, the reads constitute the input to multiple stages, namely k-mer analysis, alignment and gap closing. Reloading the reads multiple times from the parallel file system, imposes a potential I/O bottleneck for the pipeline. However, at scale we have the unique opportunity to cache the input reads and all the intermediate results onto the aggregate main memory, thus avoiding the excessive I/O and concurrent file system accesses. In order to achieve the I/O caching in a transparent way, we leverage the POSIX shared memory infrastructure and thus all the subsequent input loads are streamed through the main memory.

Figure 7.5 shows the end-to-end strong scaling performance of HipMer on the human dataset up to 23,040 Edison cores. We present this experiment in order to highlight the importance of the I/O caching. Note that the baseline concurrency is 1,920 cores; we need at least 80 Edison nodes, each with 24 cores, to fit all the data structures *and* cache the input datasets in memory ($\approx$ 5TB). The dark blue line shows the end-to-end execution time *including* the I/O, which is cached in main memory once the input reads are loaded. The ideal strong scaling is illustrated by the black line. At the concurrency of the 23,040 cores we completely assemble the human genome in 3.91 minutes and obtain a strong scaling efficiency of 48.7% relative to the baseline of 1,920 cores. In order to illustrate the effectiveness of the I/O caching, we performed the same end-to-end experiments where the input reads are loaded from the Lustre file system five times (cyan line). This experiment does not exhibit any scaling from 15,360 to 23,040 cores due to the I/O overhead, thus demonstrating that I/O caching is crucial for scaling to massive concurrencies. At the scale of 23,040 cores, the version with I/O caching is almost 2$\times$ faster than the version without this optimization.

The efficiency of the I/O (reading the input reads once) is illustrated with the magenta line. We observe that the I/O is almost a flat line across the concurrencies and yields a read bandwidth of $\approx$ 16 GB/sec (the theoretical peak of our Lustre file system is 48 GB/sec). With 80 Edison nodes we are able to saturate the available parallelism in the Lustre file system and further increasing the concurrency does not help improve the I/O performance. The red, green and orange lines show the partial execution time for (i) the k-mer analysis, (ii) contig generation and (iii) sequence alignment, scaffolding and gap closing respectively. We conclude that all the components scale up to 23,040 cores and do not impose any scalability impediments.

7.4 Performance Comparison with Other Assemblers

To compare the performance of HipMer relative to existing parallel *de novo* end-to-end genome assemblers we evaluated Ray [9, 10] (version 2.3.0) and ABySS [97] (version 1.3.6) on Edison using 960 cores. Ray required 10 hours and 46 minutes for an end-to-end run on the Human dataset. ABySS, on the other hand, took 13 hours and 26 minutes solely to get to the end of contig generation. The subsequent scaffolding steps are not distributed-memory parallel. At this concurrency on Edison, HipMer is approximately 13 times faster than Ray and *at least* 16 times faster than ABySS.

7.5 Conclusion

Next-generation short-read sequencing technology has resulted in explosive growth of sequenced DNA. However, de novo assembly has been unable to keep pace with the flood of data, due to vast computational requirements and the algorithmic complexity of assembling large-scale genomes. In this Chapter we presented the HipMer pipeline, an end-to-end high performance de novo assembler designed to scale to massive concurrencies. Our work is based on the Meraculous assembler, which has been shown to be one of the top de novo approaches in recent Assemblathon competitions.

Overall results show unprecedented performance and scalability, attaining an overall runtime of 3.91 minutes for the human DNA at 23K cores on the Cray XC30 Edison supercomputer, compared with 10.8 hours for Ray and 48 hours for the original Meraculous pipeline. Additionally, we explored performance on the grand-challenge wheat genome, which, to date, has been too large and complex for most modern de novo assemblers. Our results demonstrated impressive scalability, allowing the completed wheat assembly in just 39 minutes using 15K cores.

Chapter 8

Parallel Metagenome Assembly

Metagenomics is currently the leading technology in studying the uncultured microbial diversity and delineating the microbiome structure and function. Since the sequencing costs surpass that of Moore's Law [102], we can now access a large number of environmental samples that comprise of thousands of individual microbial genomes. Effective and accurate microbiome analysis depends on the efficiency of the assembled sequences. Assemblies of metagenomes into long contiguous sequences are not only critical for the identification of the long biosynthetic clusters [25] but are also key for enabling the discovery of novel lineages of life and viruses [29]. However, for most of these energy-related environmental samples, there is no existing reference genome, so a first step in analysis is *de novo* assembly.

As we have seen in the previous chapters, de novo assembly even for single genomes is computationally challenging. In this chapter we consider the even harder problem of metagenome assembly. In addition to the computational challenges due to the high error rates or short read lengths of sequencers, metagenome assembly is further complicated by repeated sequences across genomes, polymorphisms within species and variable frequency of the genomes within the sample. Here we present a high-performance metagenome assembly pipeline called *meta-HipMer*. Our work is based on the HipMer pipeline presented in previous chapters and maintains the goal of massive parallelism. To address the unique challenges of metagenomes we adopt many of the ideas in IDBA-UD [86], a state-of-the-art metagenome assembler that runs on shared memory systems, although we add our own innovations on the metagenome assembly.

The meta-HipMer algorithm consists of two main components: (1) the iterative contig generation and (2) the scaffolding component. The rest of this chapter is organized as follows. Section 8.1 describes the iterative contig generation, section 8.2 discusses the scaffolding module in meta-HipMer while section 8.3 details the parallelization of the meta-HipMer algorithms. Finally, section 8.4 presents experimental results where meta-HipMer is compared with other state-of-the-art metagenome assembly tools and section 8.5 concludes this chapter.

Algorithm 3 Iterative contig generation

```
1:  Input: A set of paired reads $R$
2:  Output: A set of contigs $C$
3:  $C \leftarrow \emptyset$
4:  for $k = k_{min}$ to $k_{max}$ with step $s$ do
5:      $kmerSet \leftarrow$ KMERANALYSIS$(k, R, C)$
6:      $C_k \leftarrow$ DEBRUIJNGRAPHTRAVERSAL$(kmerSet)$
7:      $C'_k \leftarrow$ BUBBLEMERGING$(C_k)$
8:      $C''_k \leftarrow$ ITERATIVEGRAPHPRUNING$(C'_k)$
9:      $R_{corrected} \leftarrow$ ERRORCORRECTION$(R, C''_k)$
10:     $C \leftarrow$ LOCALASSEMBLY$(R, C''_k)$
11:     $R \leftarrow R_{corrected}$
12: RETURN $C$
```

8.1 Iterative Contig Generation

The genomes comprising a metagenome dataset have generally variable coverage, since some species may exist with much higher frequency than others. Choosing an optimal value of k for the de Bruijn graph is therefore challenging, because there is a tradeoff in k-mer size that affects high and low frequency species differently. The iterative contig generation (Algorithm 3) aims to eliminate the quality trade-off that different k-mer sizes induce in de Bruijn graph based assemblers.

Typically, a small value of k is appropriate for low-coverage genomes since it allows sufficient number of overlapping k-mers to be found and as a result the underlying sequences can be assembled to contigs. On the other hand, a large value of k is better suited for the high-coverage genomes since a sufficient number of overlapping, long k-mers can be found and repetitive regions are disambiguated by such long k-mers. Figure 8.1 illustrates the effect of different k-mer sizes in regions with different depth of coverage. The left part of

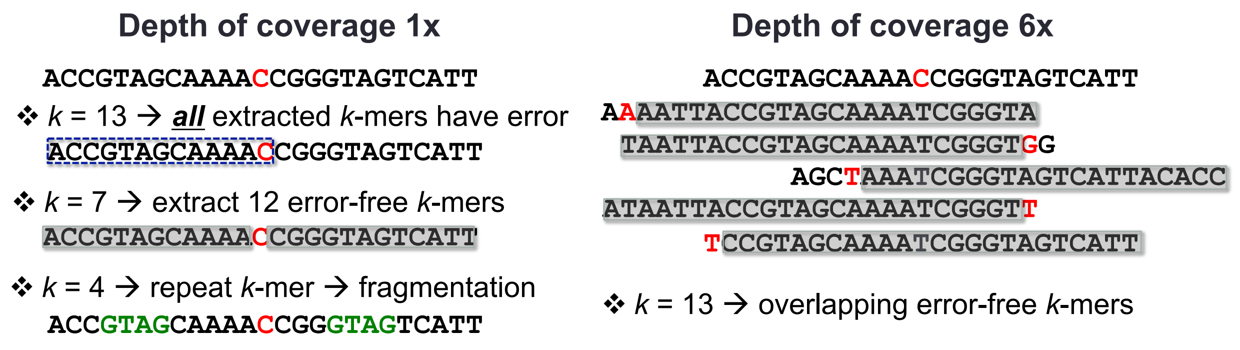

Depth of coverage 1x
ACCGTAGCAAAACCGGGTAGTCATT
k = 13 → all extracted k-mers have error
ACCGTAGCAAAACCGGGTAGTCATT
k = 7 → extract 12 error-free k-mers
ACCGTAGCAAAACCGGGTAGTCATT
k = 4 → repeat k-mer → fragmentation
ACCGTAGCAAAACCGGGTAGTCATT
Depth of coverage 6x
ACCGTAGCAAAACCGGGTAGTCATT
AAAATTACCGTAGCAAAATCGGGTA
TAATTACCGTAGCAAAATCGGGTGG
AGCTAAATCGGGTAGTCATTACACC
ATAATTACCGTAGCAAAATCGGGTT
TCCGTAGCAAAATCGGGTAGTCATT
k = 13 → overlapping error-free k-mers

Figure 8.1: Effect of k-mer size at a region with low coverage (left) and high coverage (right).

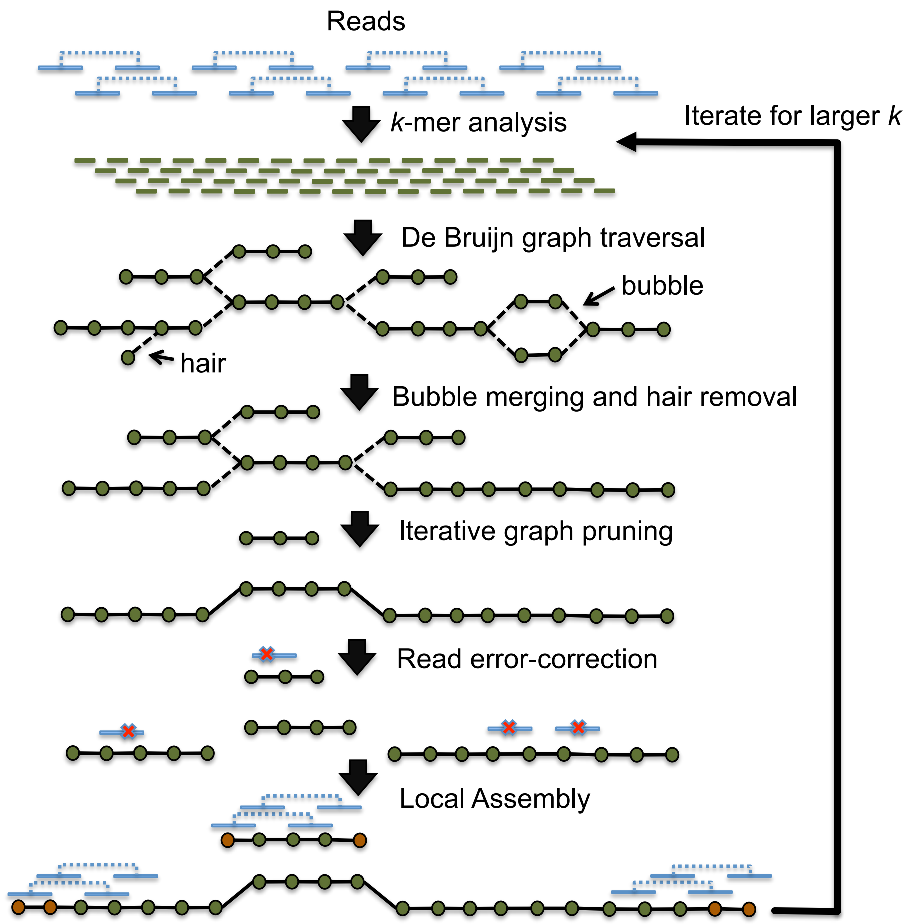


Figure 8.2: Iterative contig generation workflow in meta-HipMer.

the figure illustrates a single read with an error (red base) that covers a genomic region. By selecting $k = 13$, we are unable to extract any error-free k-mers; observe that no matter where the sliding blue window is located, it will always include the erroneous base. If we pick $k = 7$ instead, we extract in total 12 error-free k-mers within the shaded regions. By further reducing the k value down to 4, we get a repeat k-mer (`GTAG`) which will cause fragmented assembly (recall that the first k-mer analysis phase in HipMer only keeps k-mers with unique extensions, therefore the repeat k-mers will not be used in the de Bruijn graph traversal and the contig assembly will be disconnected at these repeats). In the right part of the figure we illustrate a genomic region with coverage 6× and all the reads include erroneous bases. Nevertheless, we have enough coverage and by picking $k = 13$ we can now extract sufficient number of overlapping, error-free k-mers (within the shaded regions) and we can assemble the region into a single contiguous piece.

The iterative contig generation (see Figure 8.2) starts by constructing the de Bruijn graph with a small value k and extracts the set of contigs C_k by traversing the graph. Then, k is increased by a step size s and meta-HipMer builds the corresponding de Bruijn graph of the input reads with $(k + s)$-mers while the graph is enhanced with $(k + s)$-mers extracted from the previous contig set. This iterative process is repeated until k reaches a user-specified maximum value.

The iterative analysis requires some additional steps to refine the de Bruijn graphs and therefore meta-HipMer applies a handful of transformations on these graphs (see Figure 8.2). More specifically, "bubble structures" (like the ones described in section 6.3) are merged and "hair" tips (short, dead-end dangling contigs) are removed since they are potentially created from false-positive vertices. Then, the graph is iteratively pruned in order to eliminate branches that do not comply with the coverage of the neighboring vertices; such branches are likely to be created by false-positive edges. After the pruning step, the read dataset can be optionally error-corrected by aligning the reads with the survived contigs which represent high-quality sequences. This error-correction step increases the efficiency of the subsequent iterations. Finally, the contigs that have remained in the de Bruijn graph are extended using a local assembly algorithm. In this local assembly process, the input reads are aligned onto the contigs and only the localized read sequences are used in the extension of contigs. Thus, erroneous k-mers stemming from high-coverage regions are isolated from similar k-mers in low-depth areas and the local assembly algorithm can retrieve k-mers which otherwise would be excluded from the de Bruijn graph. In the following subsections we describe the various stages of the iterative contig generation in more detail.

8.1.1 k-mer analysis with adaptive thresholding

In the first iteration of the contig generation, the k-mer analysis step chops the reads into k-mers that overlap by $k - 1$ consecutive bases, and a count is kept for each k-mer occurring more than ϵ times ($\epsilon \approx 1, 2$) in order to exclude sequencing errors. In the original HipMer algorithm which is designed for single genome assembly with uniform depth coverage, a k-mer with depth greater than ϵ was extended in the de Bruijn graph traversal process if there are no more than t_{hq} alternatives that contradict the most common extension of that k-mer. A potential problem with this approach for metagenome datasets is that genomes with high-copy number (e.g. abundance in the dataset of 1,000 or more) will typically have more than 10 alternates if the sequencing error rate is $\approx 1\%$. Therefore, setting a t_{hq} below 10 can in theory start to fragment these high-copy number genomes that are very prevalent in the dataset even though such genomes should be the easiest to assemble. On the other hand, setting a t_{hq} above 10 will fragment the low-copy number genomes. The solution is to replace the unique global threshold extension criterion with one that depends on the depth $d_{k\text{-}mer}$ of the k-mer that is being extended. In the new scheme, a k-mer is extended in the de Bruijn graph traversal if there are no more than $t_{hq} = t_{base} + e \cdot d_{k\text{-}mer}$ extensions that contradict the most common extension. Here t_{base} is a hard limit in the t_{hq} value and e is a single parameter model for the sequencing error.

In subsequent iterations of the contig generation, the k-mer analysis step takes the additional input set C of contigs from the previous iteration. These contigs are treated as

Algorithm 4 Iterative graph pruning

```
1: Input: A contig-graph $G$, $k$, $\alpha$, $\beta$, $\tau$
2: Output: A pruned graph $G_{pruned}$
3: $\tau \leftarrow 1$
4: repeat
5:    for each contig $c \in G$ do
6:       if length($c$) $\leq 2 \cdot k$ and depth($c$) $\leq \min(\tau, \beta \cdot$ neighborhood-depth($c$)) then
7:          Remove $c$ from $G$
8:    $\tau \leftarrow \tau \cdot (1 + \alpha)$
9: until $\tau \geq$ maximum contig depth of $G$
10: $G_{pruned} \leftarrow G$
```

meta-reads since they are "confidently" assembled sequences. Therefore, in addition to processing the input reads, we extract all the k-mers from the previous contigs (meta-reads) and treat them as high quality `UU` k-mers.

8.1.2 De Bruijn graph traversal

The de Bruijn graph of the k-mers stemming from the k-mer analysis is traversed in order to form *contigs*; contigs are paths in the de Bruijn graph constituted of k-mers with *unique* high quality extensions (k-mers that do not have more than $t_{base} + e \cdot d_{k\text{-}mer}$ alternatives that contradict the most common extension). These paths represent "confidently" assembled sequences and can be seen in the graph of Figure 8.2 by removing the branches (vertices with dashed incident edges) and considering the connected components in the resulting graph. Note that the vertices with dashed incident edges do not have unique high quality extensions and can be used later to discover the connectivity among the contigs.

8.1.3 Bubble merging and hair removal

A single-nucleotide polymorphism (SNP) represents a difference in a base between two genomic sequences. SNPs create similar contigs (paths in the de Bruijn graph with the same length) except one position; these contigs also have the same k-mers as extensions of their endpoints and as a result form *bubble* structures in the de Bruin graph. In this step we identify these bubbles and merge them into a single contig. Additionally, dead-end dangling contigs shorter than $2k$ nucleotides are considered *hair* and are likely to be false positive structures in the graph, hence we remove them. See Figure 8.2 for examples of a bubble and a hair contig.

8.1.4 Iterative graph pruning

The remaining graph after bubble merging and hair removal is iteratively pruned in order to eliminate branches that do not comply with the coverage of the neighboring vertices; such branches are likely to be created by false-positive edges. Algorithm 4 implements an iterative pruning strategy similar to the progressive relative depth module in IDBA-UD [86]. The reasoning behind this pruning algorithm is that:

1. Long contigs are most likely correct.
2. Wrong contigs in high-depth regions might have higher depth than correct contigs in low-depth regions; however, if these high-depth contigs are short and have relatively low depth (less than a fraction we call β of the neighbors's average depths), then they are considered erroneous and are removed.
3. Correct contigs that are short and have relatively low-depth usually have higher depths than the short wrong contigs. By employing a low-depth cutoff threshold τ that is increased progressively (by a factor we call α), we can remove such wrong contigs.

8.1.5 Read error-correction

This step is optional in meta-HipMer since the de Bruijn graph traversal implicitly handles errors (we consider only "high quality" extensions based on the adaptive thresholding). However, the error-correction will lead to more efficient subsequent iterations. The philosophy of this module is that contigs are high-quality sequences and reads that do not "fully "align onto these contigs should be corrected in order to conform to the corresponding contigs. More specifically, in the error-correction process, first we align the reads onto the generated contigs. If a read differs from the aligned contig in less than three bases, then the read is modified in order to conform to the contig sequence.

8.1.6 Local assembly

After pruning the contig graph in an iterative way, we try to extend the remaining contigs following a local assembly methodology. In this local assembly procedure, erroneous k-mers stemming from high-coverage regions are isolated from similar k-mers in low-depth areas and the local assembly algorithm can retrieve k-mers which otherwise would be excluded from the de Bruijn graph.

For the local assembly algorithm, we adapt the gap closing module of HipMer. First, we locate all the reads that can be aligned onto contigs and whose paired reads do not align onto the same contigs. By assessing the insert size of the corresponding read library and the alignment information (alignment orientation, start/end positions of alignment) we can project the unaligned reads to the "right" and "left" incidents gaps of the appropriate contigs. After assigning the unaligned reads to the appropriate gaps, we try to extend the contigs by performing appropriate "mer-walks", where only k-mers present in the reads that are projected into a gap are used. The "mer-walking" step uses a modified version of the contig generation extension algorithm applied to the projected reads. In this modification:

1. Extension bases are categorized by their quality score into four bins and extensions are accepted or rejected based on the number and quality category of extending bases. This allows for uncontested extensions of lower quality than used in the original contig generation to be accepted in local assembly.

2. The mer-size used is dynamically adjusted by "upshifting" (mer-size increased by 2) when a fork is encountered, or "downshifting" (mer-size decreased by 2) when a termination is encountered. Repeated upshifting or downshifting will occur until the data is no longer sufficient to support extension of the contig.

8.2 Scaffolding

As in the single genome case described in Chapter 6, the scaffolding algorithm attempts to link together contigs in order to create *scaffolds*, sequences of contigs that may contain gaps among them. The first step of scaffolding is to map the original reads onto the generated contigs. This mapping provides information about the relative ordering and orientations of the contigs. Once the orientations are determined, it is possible that there are gaps between pairs of contigs. We then further assess the read-to-contig mappings and locate the reads that are placed into these gaps. Ultimately, we leverage this information and close the scaffold gaps in order to get the final assembly result. We refer the reader to Chapter 6 for a more detailed description of the scaffolding algorithm.

8.3 Parallel Algorithms of meta-HipMer

In this section we detail the parallelization of the modules that are unique to the meta-HipMer pipeline, namely iterative graph pruning and read error-correction. The parallel k-mer analysis, parallel contig generation, parallel bubble merging, parallel scaffolding and parallel gap closing are described in Chapters 3, 4, 5 and 6. The parallel local assembly module leverages the same parallelization techniques as the gap closing module which is detailed in Chapter 6.

8.3.1 Parallel iterative graph pruning

The parallel version of Algorithm 4 starts with reading in parallel the contigs from the previous step along with their depths and also stores the k-mers from the k-mer analysis step in a distributed hash table. In particular, we are interested in the "fork" k-mers since they contain information regarding the connectivity of the contigs; in Figure 8.2 the vertices with dashed incident edges represent “fork” k-mers. Each one of the p processors is then assigned $1/p$ contigs, extracts the last k-mers in the two endpoints of each contig c_i, looks them up in the distributed hash table and as a result gets the contig-neighborhood $N(c_i)$ of contig c_i.

The parallel execution then proceeds in the main loop (line 4) of Algorithm 4. Each processor visits the contigs assigned to it and if a contig is both short and has relatively small depth compared to its neighborhood (line 6), then that contig should be removed from the contig graph. Note that for each contig c_i we need the neighborhood $N(c_i)$ in order to calculate the average depth of its neighborhood. At the end of the iteration, each processor updates the neighborhoods of its contigs since some of the contigs may have been removed. Then the threshold τ is increased geometrically and we proceed in the next iteration.

The parallel algorithm terminates if no contigs are pruned during an iteration. In order to detect if any contigs have been pruned from the graph: (1) every processor sets a local binary variable `pruned_flag` to 1 if *any* of its contigs have been pruned and (2) we perform an `all-reduce` operation of the `pruned_flag` variables with `max` function as argument. If the result is 1, then we know that the contig-graph has been pruned/modified and we proceed in the next iteration. On the other hand, if the result is 0 we conclude that no changes have been made in the contig-graph and the parallel algorithm terminates.

8.3.2 Parallel read error-correction

The meta-HipMer error-correction module relies on the assumption that contigs are high-quality sequences. Therefore, reads that do not “fully ”align onto a contig but have a high degree of similarity with a contig (e.g. 95%) should be corrected in order to conform to the corresponding contig sequence. These error-corrected reads will lead to more efficient subsequent iterations because they avoid the generation of erroneous k-mers and spurious edges in the de Bruijn graphs. The error-corrected reads contain also longer error-free k-mers that can be used for more accurate sequence alignment, scaffolding and gap closing.

First, we store the contig sequences from the previous step in a distributed array that can be globally accessed. For load balancing reasons, given p processors we assign contig i to the the processor $i \mod p$. Then, we align the reads onto the contigs by employing the merAligner parallel module (see Chapter 5 for more details).

After the alignment step, each processor loads $1/p$ of the reads into its memory along with the relevant alignments. Note that these alignments are local to the appropriate processors because each processor is assigned the same chunk of reads in the merAligner step. The merAligner result also includes a field that indicates if a read aligns completely to a contig; such a read should not be corrected and is considered to be error-free. If the alignment information dictates that a read r is aligned onto a contig i with a high degree of similarity (based on a user-defined threshold), then r is sent to processor $i \mod p$ that stores the contig i locally. For performance reasons, we implement message aggregation in this step.

At the last phase of the error-correction, each processor iterates over the received read sequences and modifies them in order to comply with the corresponding contig sequences.

This step of the algorithm does not involve any communication. After the error-correction stage, the reads are sent back to the processors they belong to. Finally, the processors in parallel update the input read file by replacing the old entries with the error-corrected reads.

Our parallel error-correction module can be used as a flexible stand-alone tool under different circumstances. For example the user can provide the reference sequences and in this scenario the tool performs a reference-based error-correction of the reads. If no reference sequence is available, our tool can generate a draft reference from the reads by executing the k-mer analysis and contig generation parallel modules. In such a scenario, the error-correction tool takes only the reads as input.

8.4 Experimental Results

In order to assess the quality of the meta-HipMer assemblies we performed experiments with two metagenome datasets and compared the results with two state-of-the-art metagenome assemblers, namely IDBA-UD [86] and metaSPAdes [85]. We emphasize here that both IDBA-UD and metaSPAdes are not parallelized for distributed memory systems and therefore cannot handle massive metagenome datasets.

For the first set of experiments we considered a small simulated dataset (≈ 3.2 Gbytes) with extremely uneven depth that was also used in the IDBA-UD paper. This dataset was synthesized by combining simulated reads of three species *Lactobacillus plantarum* ($\sim$3.3Mb), *Lactobacillus delbrueckii* ($\sim$1.85Mb) and *Lactobacillus reuteri F275 Kitasato* ($\sim$2Mb) from the same genus. Reads of length 100 were sampled from these three species with sequencing depth $10\times$ (genome A - low depth), $100\times$ (genome B - moderate depth) and $1,000\times$ (genome C - high depth) with 1% error rate. The simulated paired reads have an insert distance following normal distribution $\mathcal{N}(500, 50)$.

For the quality assessment of the assemblies we utilized the QUAST [41] software and we consider a few different metrics. First, we examine three reference-free metrics: (i) the number of contigs – lower is better, (ii) the total length of sequence that can be found in pieces larger that 10 kbp – higher is better – and (iii) the total length of sequence that can be found in pieces larger that 50 kbp – again, higher is better. Then we consider two metrics that take into account the reference in order to evaluate the accuracy of the assemblers: (i) the number of misassemblies (lower is better), where misassemblies are locations on an assembled contig where the left flanking sequence aligns over 1 kb away from the right flanking sequence on the reference and (ii) the number of mismatches per 100 kbp (again lower is better). Afterwards we examine the completeness of the various assemblers by considering the fraction of each reference genome that can be found in the assembly results (higher is better). Finally, we assess the contiguity of the results by providing the NGA50 metric for each genome (higher is better); NGA50 is calculated by summing all sequence lengths within a genome, starting with the longest, and observing the length that takes the sum length past 50% of the corresponding genome length.

Metric	meta-HipMer	IDBA-UD	metaSPAdes
Number of contigs	211	292	226
Total length > 10 kbp	6,511,747	6,539,057	6,540,869
Total length > 50 kbp	4,497,004	3,678,488	4,450,375
Misassemblies	4	6	13
Mismatches per 100 kbp	9.85	14.36	29.96
Genome fraction (%) A	97.57	98.5	97.94
Genome fraction (%) B	93.73	93.87	93.67
Genome fraction (%) C	94.82	95.58	94.8
NGA50 Genome A	99,607	42,608	71,507
NGA50 Genome B	48,966	56,460	61,767
NGA50 Genome C	58,776	58,782	58,451

Table 8.1: Assembly results on a metagenome dataset consisting of three species from the same genus: a low-depth genome A with sequencing depth 10×, a moderate-depth genome B with depth 100× and a high-depth genome C with coverage 1,000×.

Table 8.1 shows the assembly results of meta-HipMer, IDBA-UD and metaSPAdes on the previous dataset with highly uneven depth of coverage. In this table we use a color code in order to provide a comparison among the assemblers for each metric. Green cells indicate the best assembler in that metric and assemblers that perform within 25% of the best result. Yellow cells represent assemblers that perform within 50% of the best result on that metric. Finally, red cells stand for assemblers that do not perform within 50% of the best result on that metric. Overall, we conclude that all three metagenome assemblers perform well on this dataset with extremely uneven depths of coverage; all of them are specially designed to take into account the idiosyncrasies of metagenomes. In regard to accuracy, meta-HipMer is the most accurate and exhibits fewer misassemblies and fewer mismatches than IDBA-UD and metaSPAdes. Such behavior is in alignment with the empirical behavior of HipMer compared to other tools: meta-HipMer's de Bruijn graph traversal is conservative and k-mers are extended only if they have unique high quality extensions (this traversal feature is unique to HipMer/meta-HipMer pipeline). By examining the genome fractions that could be assembled we conclude that all three assemblers have similar performance and perform well regarding completeness. The assemblers are able to assemble $\approx$ 98% of the low-coverage genome A, $\approx$ 94% of the moderate-coverage genome B and $\approx$ 95% of the high-coverage genome C. Finally, meta-HipMer exhibits significantly better contiguity (metric NGA50) in the low-coverage genome A compared to the other two competing assemblers, while it has similar performance with respect to the contiguity of genomes B and C.

For the second set of experiments we considered a larger metagenome dataset ($\approx$ 110 Gbytes) from Joint Genome Institute. This dataset represents a 26 member synthetic community of bacteria from real sequencing data and the members have various depths of cover-

Metric	**meta-HipMer**	**metaSPAdes**
Number of contigs	2,204	5,100
Total length $>$ 10 kbp	93,228,737	92,152,728
Total length $>$ 50 kbp	75,061,366	77,292,121
Misassemblies	63	134
Mismatches per 100 kbp	9.79	77.05
Genome fraction (%)	91.805	91.10

Table 8.2: Assembly results on a synthetic community consisting of 26 members with different abundances.

age. In this experiment we compare only meta-HipMer and metaSPAdes and the results are summarized at Table 8.2. Again we conclude that both assemblers perform similarly well in regard to completeness ($\approx$ 92% of the underlying genomic sequences are assembled) and contiguity (similar number of total sequence length in pieces larger than 10 kbp and 50 kbp). However, meta-HipMer is more accurate than metaSPAdes on this dataset; it has almost 2$\times$ fewer misassemblies and almost 8$\times$ fewer mismatches per 100 kbp. This dataset also reveals the limitations of metaSPAdes and all other metagenome assemblers that do not employ distributed memory parallelism. MetaSPAdes required **11.5 hours** on a 32 core machine with 500 GB of memory. On the other hand, the meta-HipMer assembly was performed on 1,920 cores (80 nodes) of Edison and it required **32 minutes**, being approximately 22$\times$ faster than metaSPAdes. Given that real metagenome datasets can be orders of magnitude larger than this mock community, we conclude that shared memory assemblers are incapable of dealing with real-world, massive and complex metagenomes. On the contrary, meta-HipMer is a high-quality, end-to-end parallelized pipeline that can assemble challenging metagenomes on modern distributed memory systems in a time and memory efficient manner.

8.5 Conclusion

In this Chapter we presented meta-HipMer, a high-quality metagenome assembly pipeline designed to scale to massive concurrencies. The meta-HipMer pipeline deals efficiently with repeated sequences across genomes, polymorphisms within species and variable frequency of the genomes within the sample by employing an iterative contig generation algorithm. All the components of meta-HipMer are parallelized and can be executed on distributed memory systems. Therefore, meta-HipMer alleviates the need for large shared memory, specialized machines. Empirical results demonstrate that our work outperforms state-of-the-art tools in both quality and performance and it enables the high-quality assembly of massive metagenomes datasets which are prohibitive for all the other available tools.

Metagenomic sequencing is growing at a rapid pace. New phyla of bacteria are still being discovered by DNA sequencing from diverse environments, allowing genomes to be sequenced for microbes that cannot yet be cultured in the laboratory [14, 91]. Routinely assembly of metagenomes, without the limitation of large memory single node systems, promises to discover novel microbial genomes and reveal deep new branches in the tree of life [48, 104]. Often, the properties of a microbe can be inferred from its genes, suggesting novel cultivation (or antibiotic) strategies. An unusual feature of microbial genomes is their high degree of plasticity, and even closely related "species" can differ by the addition and/or subtraction of genes exchanges with other species in their environment. Metagenome assembly allows these novel genomic combinations to be identified and characterized even in the absence of cultivated species. In a human microbiome context, routine metagenome assembly will contribute to the promotion of healthy microbiomes and the development of new treatments for microbiome-related disease. We envision the bioinformatics community adopting the meta-HipMer pipeline and assembling such grand-challenge, previously intractable metagenome datasets.

Chapter 9

Architectural Analysis of Core Operations

In this chapter we present an architectural analysis of critical operations encountered in our pipeline. First, we investigate communication aspects that are stressed during the pipeline and we illustrate the impact of the communication infrastructure on the parallel contig generation and sequence alignment algorithms presented in Chapters 4 and 5. We develop micro-benchmarks to assess the behavior of irregular access patterns at different scales and we compare the performance of the micro-benchmarks with the empirical performance of the real application modules. Finally, we present scalability limitations in the I/O infrastructure and highlight the necessity for scalable, parallel I/O in modern file systems in order to accommodate genome assembly pipelines that deal with massive datasets. For the results presented in this chapter we utilized the Edison supercomputer, but the methodology is general and applicable to any other system.

9.1 Communication Infrastructure Analysis

The communication system of Edison is the Cray Aries high-speed interconnect with Dragonfly topology [30] and we investigate its efficiency using three metrics: the bandwidth, the global atomics latency and one-sided remote read (get) latency.

9.1.1 Bandwidth

In this section we focus on the *bisection bandwidth* [44] of the underlying system: the available bandwidth between two partitions of the system where each partition has the same number of nodes. Parallel operation with all-to-all communication pattern typically stress the bisection bandwidth of the system and are prevalent in the parallel algorithms presented in this dissertation. In order to produce a simple micro-benchmark, we will use a synchronous all-to-all in which all threads start together, send and receive the same volume, and wait for

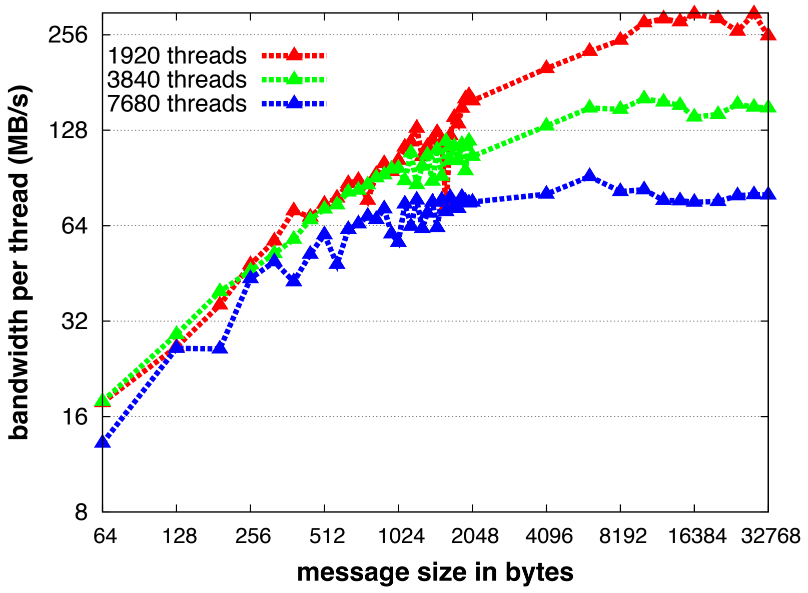


Figure 9.1: Achieved per-thread bandwidth at different concurrencies using different message sizes. The communication pattern is an all-to-all exchange implemented with remote get operations. Both axes are in logarithmic scale.

others to finish. This is the worst case in terms of stressing the bisection bandwidth of the machine, but because of load imbalance in communication and different patterns of remote vs local accesses across machines, the HipMer code uses asynchronous all-to-all communication. The entire k-mer analysis step is built around the all-to-all exchange of k-mers and the ubiquitous distributed hash table construction algorithm includes an all-to-all exchange of the inserted objects.

The Cray Aries interconnect on Edison has the physical infrastructure to provide at most 23.7 TB/s global bisection bandwidth. However, the Edison system is shared among multiple users and realistic use case scenarios do not involve the entire supercomputer. As a result, the attainable bandwidth at different scales varies from the full-system bisection bandwidth. In order to evaluate the bandwidth we perform the following experiment: the processors perform an all-to-all communication operation implemented with remote get primitives. We execute multiple experiments by varying the message size and the results are exhibited at Figure 9.1 for three different concurrencies (1,290, 3,840 and 7,680 threads). All the experiments are performed in the same allocation of 7,680 cores (320 Edison nodes). The x axis shows in log scale the message size in bytes while the y axis shows the attained *per thread* bandwidth in MB/s. As we increase the message size, the achieved per thread bandwidth is also increased and after some point it reaches a plateau. At the concurrency of 1,920 threads the peak *aggregate* bisection attained bandwidth is 559 GB/s, while at 3,840 threads and 7,680 threads we achieve a total bandwidth of 567 GB/s and 601 GB/s respectively.

Now we describe a proxy benchmark that is closer to the actual implementation of the all-

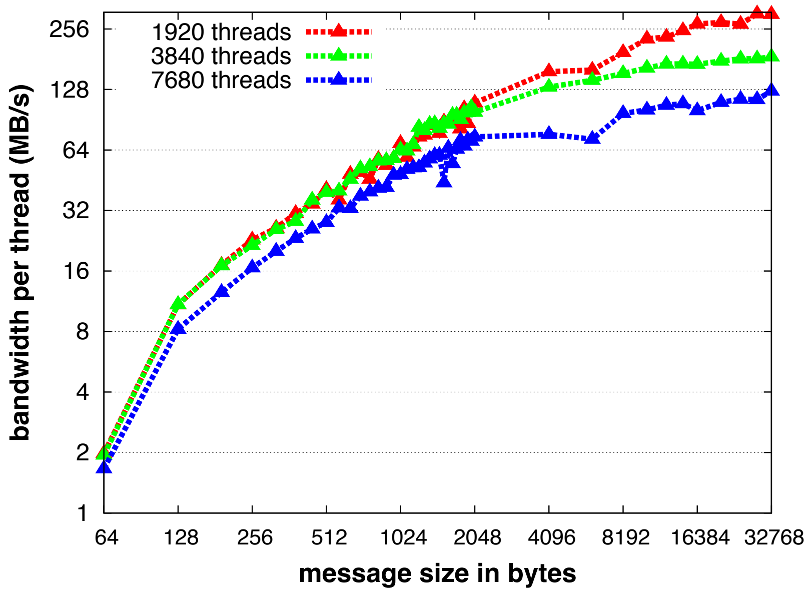


Figure 9.2: Proxy micro-benchmark for the all-to-all exchange in the distributed hash table construction with dynamic message aggregation. The y axis shows the attained effective bandwidth per thread while the x axis shows the message size.

to-all operation into the distributed hash table construction phase (see section 4.1 for more details). This micro-benchmark naturally provides a tighter upper bound on the performance of the real all-to-all exchange at the application level. In this micro-benchmark:

1. Every processor picks a random processor id p'
2. Performs a remote atomic `fetch_and_add()` on an integer variable that belongs to p'
3. Performs an aggregate remote transfer of size S to processor p'
4. Repeat steps 1-3 multiple times

The atomic `fetch_and_add()` corresponds to the required action that ensures atomicity in the data exchange (i.e. avoid overwrite of the data by multiple processors as explained in section 4.1).

Figure 9.2 illustrates the results of this micro-benchmark for three different concurrencies: 1,920, 3,840 and 7,680 threads. The x axis shows the message size S and the y axis shows the attained effective bandwidth per thread. First we observe that the results illustrate similar behavior to the previous bandwidth benchmark. Figure 9.3 shows the effective *aggregate* bisection bandwidth for the proxy micro-benchmark (red bars) and the bandwidth micro-benchmark (green bars) at the concurrency of 3,840 cores and we see that for large messages the differences are negligible. The proxy micro-benchmark also dictates the minimum required aggregate message size to saturate the available bandwidth at each concurrency. Note though that the dynamic message aggregation detailed in section 4.1 requires $S \times P$ times more memory per thread (given P threads in total). Therefore, we should tune the S parameter at each concurrency according to the available memory.

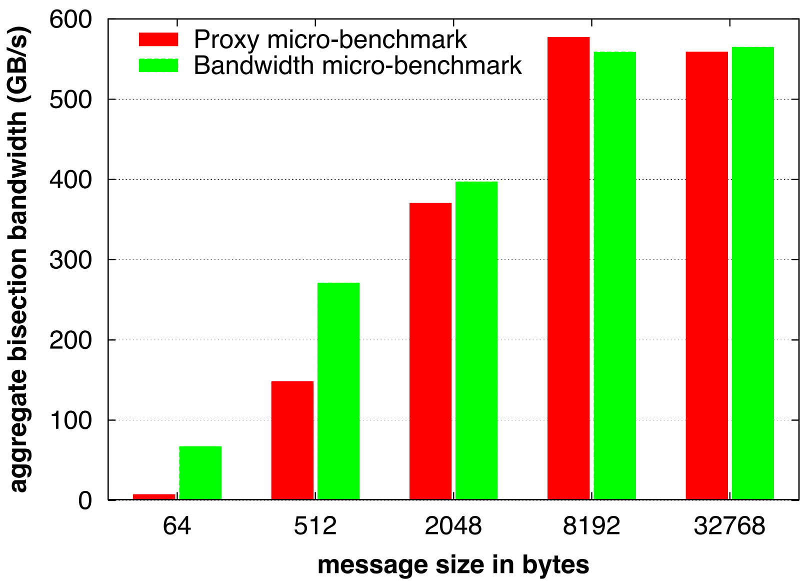


Figure 9.3: Effective aggregate bisection bandwidth for the proxy micro-benchmark (red bars) and the bandwidth micro-benchmark (green bars) at the concurrency of 3,840 cores.

Concurrency (number of threads)	1,920	3,840	7,960
Effective bandwidth of proxy benchmark (GB/s)	300	533	543
Effective bandwidth of application code (GB/s)	87	112	109

Table 9.1: Comparison of communication proxy benchmark and application code that performs the all-to-all exchange. The message size is 6,400 bytes.

Table 9.1 compares the aggregate bisection bandwidth performance of the proxy benchmark to the all-to-all exchange in the distributed hash table construction phase. The message size in these experiments is 6,400 bytes. We emphasize that the proxy benchmark captures only the communication cost (e.g. it ignores buffer copies, hashing overheads, skewed data distributions, local computations) and consequently yields a loose upper bound in performance. Nevertheless, the empirical performance of the application is always within 5× of the upper bound provided by the proxy benchmark. By further investigating the application results at the concurrency of 3,840 cores, we found that 47% of the execution time is required for local computations. In other words, the effective bisection bandwidth of the application considering only the communication operations is 219 GB/s, which is within 2.5× of the upper bound provided by the proxy micro-benchmark.

9.1.2 Global atomics

Another important aspect of the communication interconnect that is stressed throughout the pipeline is the efficiency of global atomics. In particular, we focus on the global atomic

Concurrency (number of threads)	1,920	3,840	7,960
Latency (μs) to visit a vertex in the proxy benchmark	9.8	14.78	21.6
Latency (μs) to visit a vertex in the application	24.2	32.21	43.9

Table 9.2: Comparison of proxy benchmark and application code that performs the parallel de Bruijn graph traversal.

`compare_and_swap()` that is crucial for the parallel de Bruijn graph traversal described in section 4.2. The proxy benchmark that models the communication and synchronization behavior of the graph traversal algorithm is the following:

1. Every processor fetches a random entry from the distributed hash table
2. Performs a global atomic `compare_and_swap()` on an integer variable of the previously selected entry
3. Repeat steps 1-2 multiple times

This proxy micro-benchmark represents the minimum required latency overhead in order to visit a vertex in the distributed de Bruijn graph traversal. It captures only the communication cost of this primitive operation and it does not consider any overheads due to the high-level synchronization protocol, hashing and local computations. For example, the results at Figure 4.10(a) in Chapter 4 show that at the concurrency of 7,680 cores almost 45% of the time is spent in the synchronization protocol. Therefore, the proxy micro-benchmark provides a loose upper bound on the performance of the parallel graph traversal. Table 9.2 compares the performance of the proxy benchmark and the parallel graph traversal code described in section 4.2. We conclude that the empirical performance of the application is always within roughly 2$\times$ and 3$\times$ of the upper bound provided by the proxy communication benchmark. The results also indicate that global atomics which take advantage of hardware support will speedup significantly this latency-sensitive parallel algorithm.

9.1.3 Latency

Here we examine the latency of the one-sided get operation at scale. This primitive is used to implement the lookup functionality in the distributed hash table and therefore is the limiting factor in most of the parallel algorithms with irregular accesses. For example, the sequence alignment algorithm described in Chapter 5 spends most of its execution time in distributed seed index (hash table) lookups.

Figure 9.4 illustrates the remote get latency for different message sizes at three different concurrencies. We are particularly interested in the regime of small messages sizes because the hash table entries typically have size up to 128 bytes; for such message sizes the latency varies between 4 to 6 microseconds. Table 9.3 compares the latency of a get operation with the latency of a lookup in the distributed hash table used in the alignment algorithm (the message size is 64 bytes). Note that the lookup operation also includes the hash computation, overhead to validate that the fetched entry is the one with the requested key and potential

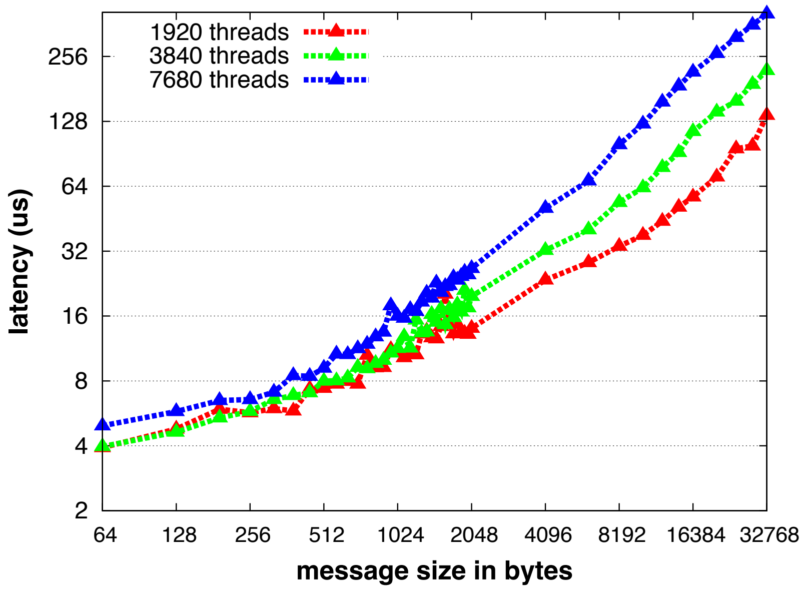


Figure 9.4: Latency to fetch an entry from the distributed hash table. The y axis shows the latency in microseconds while the x axis shows the message size. Both axes are in logarithmic scale.

Concurrency (number of threads)	1,920	3,840	7,960
Latency of remote get (μs)	3.9	4.0	5.0
Latency of lookup in the distributed hash table (μs)	11.7	11.4	9.81

Table 9.3: Comparison of get latency and latency of lookup in the distributed hash table. The message size is 64 bytes.

overheads to follow remote chain-pointers at hash table buckets with conflicts. Nevertheless, the empirical performance of the lookup is always within 3× of the latency of a single get operation at all concurrencies. The fact that the lookup latency remains almost constant at all concurrencies justifies the efficient scaling of the parallel sequence alignment. As we increase the number of cores, proportionally fewer lookups are executed on the critical path and since they have constant latency the total execution time decreases proportionally with the number of cores. The same scaling argument applies to all key operations in our parallel algorithms that involve irregular accesses in the distributed hash tables.

The current implementation is already scalable given the above latency comparison, but the absolute performance for a given machine size could likely be improved with some form of message aggregation and computation/communication overlap.

9.2 I/O Scalability Analysis

The experimental results in section 7.3.1 show that the I/O time is almost constant across all concurrencies and we conclude that 80 Edison nodes are sufficient to saturate the available Lustre file system bandwidth. In aggregate we achieve a read bandwidth of $\approx$ 16 GB/sec, while the theoretical peak of our Lustre file system is 48 GB/sec. As a result, further increasing the concurrency does not help improve the I/O performance and this limitation imposes a scalability impediment for the whole pipeline, even though at 1,920 cores only 3% of the total execution time is required for I/O. In this section we analyze this I/O limitation starting with a simple analysis analogous to Amdahl's law [92].

Let T_{comp} be the time required for computations in the pipeline and $T_{I/O}$ be the time required to read the input dataset. If we assume perfect scalability of both computation and I/O, the total execution time of the pipeline $T(p, p_{I/O})$ as a function of the concurrency p and the I/O parallelism $p_{I/O}$ can be written as:

$$T(p, p_{I/O}) = \frac{T_{comp}}{p} + \frac{T_{I/O}}{p_{I/O}} \tag{9.1}$$

For this I/O analysis we assume perfect scalability of the computation and take the limit as p goes to infinity:

$$T(\infty, p_{I/O}) = \frac{T_{I/O}}{p_{I/O}} \tag{9.2}$$

Therefore, the maximum attainable *relative* speedup $S_{relative}(p)$ for the pipeline over an execution with p processors is:

$$S_{relative}(p) = \frac{T(p, p_{I/O})}{T(\infty, p_{I/O})} = \frac{\frac{T_{comp}}{p} + \frac{T_{I/O}}{p_{I/O}}}{\frac{T_{I/O}}{p_{I/O}}} = \frac{T_{comp}}{p}\frac{p_{I/O}}{T_{I/O}} + 1 \tag{9.3}$$

As we described in section 7.3.1, the human genome can be assembled in 22.8 minutes using 1,920 cores while reading the input file requires 0.67 minutes. By setting $\frac{T_{I/O}}{p_{I/O}} = 0.67$ and $\frac{T_{comp}}{p} = 22.8$ in equation 9.3 we get: $S_{relative}(1{,}920)$=35. In other words, even though the full-system Edison is 70$\times$ larger than the partition of 1,920 cores, the theoretical performance speedup is upper-bounded by 35$\times$ due to the I/O limitation.

9.3 Conclusion

In this chapter we presented an architectural analysis of core operations encountered in our pipeline. In particular we developed micro-benchmarks to quantify the bisection bandwidth, the network latency and the efficiency of remote atomics. We then compared the performance of the micro-benchmarks to the performance of application-level operations that stress these communication aspects and showed that our implementations perform reasonably well. Finally, we analyzed the scalability bottleneck due to I/O limitations and emphasized the necessity of scalable I/O for our data intensive workloads in a strong scaling scenario.

Chapter 10

Related Work

Genome analysis pipelines have attracted a lot of attention over the last decade due to the exponential increase of genomic data. As a result, there is a wide body of work related to the themes in this dissertation. In this chapter, we review some of the work that has been done in the areas of genome assembly, sequence alignment and metagenome assembly. This is not an exhaustive review of the work, but we discuss a subset of the related work in order to provide some context to our own work and highlight the necessity of high-quality, scalable pipelines for genome analysis.

10.1 De novo genome assembly

As there are many *de novo* genome assemblers and assessment of the quality of these is well beyond the scope of this thesis, we refer the reader to the work of the Assemblathon II [13] as an example of why Meraculous [19] was chosen to be scaled, optimized and re-implemented as HipMer. Here we primarily refer to parallel assemblers with the potential for strong scaling on large genomes (such as plant, mammalian and metagenomes).

Ray [9, 10] is an end-to-end parallel de novo genome assembler that utilizes MPI and exhibits strong scaling. It can produce scaffolds directly from raw sequencing reads and produces timing logs for every stage. As shown in Section 7.4 Ray is approximately 13× slower than HipMer for the human data set on 960 cores. The performance difference may be due to the communication or algorithmic differences but at least part is attributable to the lack of parallel I/O.

ABySS [97] was the first de novo assembler written in MPI that also exhibits strong scaling. Unfortunately only the first assembly step of contig generation is fully parallelized with MPI and the subsequent scaffolding steps must be performed on a single shared memory node. As shown in Section 7.4 ABySS' contig generation phase is approximately 16× slower than HipMer's entire end-to-end solution for the human data set on 960 cores.

PASHA [72] is another partly MPI based de Bruijn graph assembler, though not all steps are fully parallelized as its algorithm, like ABySS, requires a large memory single node for the last scaffolding stages. The PASHA authors do claim over 2× speedup over ABySS on the same hardware.

YAGA [52] is a parallel distributed-memory that is shown to be scalable except for its I/O. HipMer employs efficient, parallel I/O so is expected to achieve end-to-end performance scalability. Also, the YAGA assembler was designed in an era when the short reads were extremely short and therefore its run-time will be much slower for current high throughput sequencing systems.

SWAP [79] is a relatively new parallelized MPI based de Bruijn assembler that has been shown to assemble contigs for the human genome and performs strong scaling up to about one thousand cores. However, SWAP does not perform any of the scaffolding steps, and is therefore not an end-to-end *de novo* solution. Additionally, the peak memory usage of SWAP is much higher than HipMer, as it does not leverage Bloom filters.

There are several other shared memory assemblers that produce high quality assemblies, including ALLPATHS-LG [38] (pthreads/OpenMP parallelism depending on the stage), SOAPdenovo [67] (pthreads), DiscovarDenovo [53] (pthreads) and SPAdes [7] (pthreads), but unfortunately each of these requires a large memory node and we were unsuccessful at running experiments using our datasets on a system containing 512GB of RAM due to lack of memory. This shows the importance of strong scaling distributed memory solutions when assembling large genomes.

In regard to k-mer analysis solutions, Jellyfish [75] is a shared-memory k-mer counting software. Khmer [21] is a pre-publication software for k-mer counting that uses Bloom filters but is not designed for distributed memory machines. These approaches are more restrictive as they are limited by the concurrency and memory capacity of the shared-memory node. Kmernator [27] is distributed k-mer counter but it consumes more memory than our k-mer analysis algorithm since it does not use Bloom filters.

10.2 Sequence alignment

A thorough survey of sequence aligners is beyond the scope of this thesis and we refer the reader to [43, 65, 32, 93, 46, 94]. We primarily focus on parallel sequence mapping tools and relevant methods in this section.

CUSHAW2 [71], BWA [64], BWA-mem [63], Bowtie2 [60], SNAP [106], SOAP [66] and GSNAP [105] are mapping tools that employ shared memory parallelism during the aligning phase. However, these approaches are more restrictive as they are limited by the concurrency and memory capacity of the shared-memory node. CUSHAW2-GPU [70] and SOAP3-dp [73] are short read aligners that leverage GPU power on a single compute node. pMap [88] is an MPI-based tool used to parallelize existing short sequence mapping tools (like the ones mentioned above) by partitioning the reads and distributing the work among the processors. However, pMap does not leverage any parallelism during the index table construction and therefore a serialization bottleneck is introduced in the mapping pipeline. PBWA [87] employs MPI in order to execute BWA on distributed memory machines, however the index table construction and its replication are serial processes. Also, the sequence distribution is done by a single master process. Therefore, pBWA suffers from the same limitations as pMap. Menon et al. [80] parallelize the genome indexing with MapReduce, however the scaling they obtain is poor.

Bozdag et al. [12] evaluate different methods of distributed memory parallelization of a mapping pipeline. These methods fall basically in three categories: (i) partitioning the reads only, (ii) partitioning the genome (and consequently the index table) and (iii) hybrid method of (i) and (ii) that partitions both reads and the genome (and the index table). One conclusion of this study is that method (i) suffers from the serialized index table construction, method (ii) does not scale in the mapping phase, and regarding method (iii), even though it exhibits improved scalability, its scaling is not close to linear. The main reason is that in the hybrid method, the index table creation is parallelized among subgroups of processes and the reads are also partitioned among subgroups of processes. Therefore, the hybrid method does not exploit the highest possible level of concurrency. Our work does fully parallelize the index table creation and partitions the reads using all available processors.

pFANGS [83] also tries to parallelize both the index table construction and the alignment phase. It distributes the index table among the processors but the processors cannot look up the distributed index in arbitrary locations. Therefore, the index lookup tasks are localized first, then an all-to-all personalized communication step is performed, the local lookups take place, and finally the lookup results are redistributed such that they are placed with the relevant queries (this redistribution is done with all-to-all personalized communication). The authors identify that the communication becomes a bottleneck because of the all-to-all communication and therefore they divide the processes in disjoint subgroups where each subgroup works independently by creating its own copy of the index table. However, in this approach the scaling of the index table construction is limited by the size of each subgroup.

Orion [74] is an improvement over mpiBLAST [22] and scales the sequence matching with fine-grained parallelization. However, Orion uses mpiBLAST's `mpiformatdb` tool to format and partition the database and this process is serial.

Finally, González-Domínguez et al. [39] describe a parallel implementation in a PGAS model with UPC++ [109] of the CUSHAW3 [69] aligner for multi-core clusters with improved scalability compared to pMap. Their goal is to parallelize aligners that work with index data structures which typically fit in the main memory of one node and therefore they do not parallelize the seed index construction phase.

10.3 Metagenome assembly

The metagenome assembly is an active research field where many algorithms have emerged. Among them Genovo [61], IDBA-UD [86], Bambus 2 [57], metaSPAdes [85], metaVelvet [84], MEGAHIT [62] and Omega [42] produce high-quality metagenome assemblies on a variety of datasets. However, all these assemblers do not employ distributed memory parallelism and hence their scalability is limited to the concurrency and memory of a single node.

The results at Chapter 8 illustrate the benefits of our distributed memory solution. One exception among the metagenome assemblers is Ray-Meta [10] which is implemented in MPI and employs distributed memory parallelism. Nevertheless, Ray-Meta has been shown to produce worse assemblies especially in regard to contiguity and completeness compared to metaSPAdes [85] on complex metagenome datasets.

The number of metagenomic datasets in public databases is now doubling every 11 months [100], projecting to grow to nearly a million metagenomes datasets in 2020. We conclude therefore that shared-memory metagenome assemblers cannot keep up with current and future grand-challenge metagenomes. The meta-HipMer assembly pipeline presented in this dissertation constitutes the first scalable, high-quality metagenome assembler.

Chapter 11

Conclusions

In this dissertation we presented HipMer, the first end-to-end highly scalable, high-quality de novo genome assembler, demonstrated to scale efficiently on tens of thousands of cores. HipMer is two orders of magnitude faster than the original Meraculous code and at least an order of magnitude faster than other assemblers, including those with incomplete pipelines and lower quality. Parts of the HipMer pipeline were used in the first whole-genome assembly of the grand-challenge wheat genome [20]. HipMer is so fast, that by using just 17% of Edison's cores, we could assemble 90 Tbases/day, or all of the 5,400 Tbases in the Sequence Read Archive [101] in just 2 months. Also, the HipMer technology makes it possible to improve assembly quality by running tuning parameter sweeps that were previously prohibitive in terms of computation.

We also presented meta-HipMer, the first scalable, distributed memory metagenome assembler. We showed that meta-HipMer produces assemblies that are competitive or better in quality than those of previous state-of-the-art metagenome assemblers, but at least an order of magnitude faster, because our pipeline can scale efficiently to distributed memory architectures. But most importantly, meta-HipMer is not limited by the concurrency and memory of a single node and thus it can handle multi-terabyte metagenome datasets that other tools are incapable of dealing with. Due to the unforeseen drop in sequencing costs, it is now routine for sequence datasets from a single sample to exceed one terabyte in size and expected to grow by at least an order of magnitude by 2020.

Obtaining these scalable pipelines required several new parallel algorithms and distributed data structures which take advantage of a global address space model of computation on distributed memory hardware, remote atomic memory operations and novel synchronization protocols. Additionally, we developed runtime support to reduce communication cost through dynamic message aggregation, and statistical algorithms that reduced communication through locality aware hashing schemes. We showed that the building block of HipMer can be repurposed for related problems such as the metagenome assembly, and that high-performance distributed hash tables, with various optimizations constitute a powerful abstraction for this type of irregular data analysis problems.

We believe our results will be important both in the application of assembly to health and environmental applications and in providing a conceptual framework for scalable genome analysis algorithms beyond those presented here. The code for HipMer is open source and can be downloaded at: `https://sourceforge.net/projects/hipmer/`.

Bibliography

[1] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. "Mergeable summaries". In: *ACM Transactions on Database Systems (TODS)* 38.4 (2013), p. 26.

[2] AA Althani, HE Marei, WS Hamdi, GK Nasrallah, ME El Zowalaty, S Al Khodor, M Al-Asmakh, H Abdel-Aziz, and C Cenciarelli. "Human Microbiome and its Association With Health and Diseases". In: *J Cell Physiol* (Dec. 11, 2015).

[3] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. "Basic local alignment search tool". In: *Journal of molecular biology* 215.3 (1990), pp. 403–410.

[4] Austin Appleby. *Murmurhash.* `https://sites.google.com/site/murmurhash/`. 2011.

[5] David A Bader, David R Helman, and Joseph JáJá. "Practical parallel algorithms for personalized communication and integer sorting". In: *Journal of Experimental Algorithmics (JEA)* 1 (1996), p. 3.

[6] Hari Balakrishnan, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. "Looking up data in P2P systems". In: *Communications of the ACM* 46.2 (2003), pp. 43–48.

[7] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A. Gurevich, Mikhail Dvorkin, Alexander S. Kulikov, Valery M. Lesin, Sergey I. Nikolenko, Son Pham, Andrey D. Prjibelski, Alexey V. Pyshkin, Alexander V. Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, and Pavel A. Pevzner. "SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing". In: *J Comput Biol.* 19.5 (May 2012), pp. 455–477.

[8] Burton H Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[9] Sébastien Boisvert, François Laviolette, and Jacques Corbeil. "Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies". In: *Journal of Computational Biology* 17.11 (2010), pp. 1519–1533.

[10] Sébastien Boisvert, Frédéric Raymond, Élénie Godzaridis, François Laviolette, Jacques Corbeil, et al. "Ray Meta: scalable de novo metagenome assembly and profiling". In: *Genome Biology* 13.R122 (2012).

[11] Dan Bonachea. *GASNet Specification, v1.1*. `http://gasnet.lbl.gov/CSD-02-1207.pdf`. 2002.

[12] Doruk Bozdag, Catalin C Barbacioru, and Umit V Catalyurek. "Parallel short sequence mapping for high throughput genome sequencing". In: *IPDPS*. IEEE. 2009.

[13] Keith R Bradnam, Joseph N Fass, Anton Alexandrov, Paul Baranay, Michael Bechner, et al. "Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species". In: *GigaScience* 2.1 (2013), pp. 1–31.

[14] CT Brown, LA Hug, BC Thomas, I Sharon, CJ Castelle, A Singh, MJ Wilkins, KC Wrighton, KH Williams, and Banfield JF. "Unusual biology across a group comprising more than 15% of domain Bacteria". In: *Nature* 523 (7559 July 9, 2015), pp. 208–11.

[15] Monika ten Bruggencate and Duncan Roweth. "Dmapp-an api for one-sided program models on baker systems". In: *Cray User Group Conference*. 2010.

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

[17] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. "Introducing OpenSHMEM: SHMEM for the PGAS community". In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM. 2010, p. 2.

[18] JA Chapman, I Ho, E Goltsman, and DS Rokhsar. "Meraculous2: fast accurate short-read assembly of large polymorphic genomes." In: *PLOS* (Submitted, 2016).

[19] Jarrod A. Chapman, Isaac Ho, Sirisha Sunkara, Shujun Luo, Gary P. Schroth, and Daniel S. Rokhsar. "Meraculous: De Novo Genome Assembly with Short Paired-End Reads". In: *PLoS ONE* 6.8 (Aug. 2011), e23501.

[20] Jarrod Chapman, Martin Mascher, Aydın Buluç, Kerrie Barry, Evangelos Georganas, Adam Session, Veronika Strnadova, Jerry Jenkins, Sunish Sehgal, Leonid Oliker, Jeremy Schmutz, Katherine Yelick, Uwe Scholz, Robbie Waugh, Jesse Poland, Gary Muehlbauer, Nils Stein, and Daniel Rokhsar. "A whole-genome shotgun approach for assembling and anchoring the hexaploid bread wheat genome". In: *Genome Biology* 16.26 (2015). DOI: `doi:10.1186/s13059-015-0582-8`.

[21] Michael R. Crusoe, Greg Edvenson, Jordan Fish, Adina Howe, Luiz Irber, et al. *khmer – k-mer counting & filtering FTW*. `https://github.com/ctb/khmer`. 2014.

[22] Aaron Darling, Lucas Carey, and Wu-chun Feng. "The design, implementation, and evaluation of mpiBLAST". In: *Proceedings of ClusterWorld* 2003 (2003).

[23] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. "Frequency Estimation of Internet Packet Streams with Limited Space". In: *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*. 2002, pp. 348–360.

[24] James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. "An implementation and evaluation of the MPI 3.0 one-sided communication interface". In: *Concurrency and Computation: Practice and Experience* (2016).

[25] Mohamed S Donia, Peter Cimermancic, Christopher J Schulze, Laura C Wieland Brown, John Martin, Makedonka Mitreva, Jon Clardy, Roger G Linington, and Michael A Fischbach. "A systematic analysis of biosynthetic gene clusters in the human microbiome reveals a common family of antibiotics". In: *Cell* 158.6 (2014), pp. 1402–1414.

[26] *Edison supercomputer*. `http://www.nersc.gov/users/computational-systems/edison/`. Accessed: 2016-07-18.

[27] Rob Egan. *Kmernator: An MPI Toolkit for large scale genomic analysis*. `https://github.com/JGI-Bioinformatics/Kmernator`. 2014.

[28] Carla Schlatter Ellis. "Concurrency in linear hashing". In: *ACM Transactions on Database Systems (TODS)* 12.2 (1987), pp. 195–217.

[29] Emiley A Eloe-Fadrosh, Natalia N Ivanova, Tanja Woyke, and Nikos C Kyrpides. "Metagenomics uncovers gaps in amplicon-based detection of microbial diversity". In: *Nature Microbiology* 1 (2016), p. 15032.

[30] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, James Reinhard, et al. "Cray cascade: a scalable HPC system based on a Dragonfly network". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press. 2012, p. 103.

[31] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm". In: *DMTCS Proceedings* (2008). ISSN: 1365-8050.

[32] Nuno A Fonseca, Johan Rung, Alvis Brazma, and John C Marioni. "Tools for mapping high-throughput sequencing data". In: *Bioinformatics* (2012), bts605.

[33] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. "HipMer: An Extreme-Scale De Novo Genome Assembler". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. 2015.

[34] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. "merAligner: A Fully Parallel Sequence Aligner". In: *Proceedings of the IPDPS*. 2015.

[35] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. "Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. 2014.

[36] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. "Enabling highly-scalable remote memory access programming with MPI-3 one sided". In: *2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE. 2013, pp. 1–12.

[37] Tarek El-Ghazawi and Lauren Smith. "UPC: unified parallel C". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 27.

[38] S Gnerre, I MacCallum, D Przybylski, F Ribeiro, J Burton, B Walker, T Sharpe, G Hall, T Shea, S Sykes, A Berlin, D Aird, M Costello, R Daza, L Williams, R Nicol, A Gnirke, C Nusbaum, ES Lander, and DB Jaffe. "High-quality draft assemblies of mammalian genomes from massively parallel sequence data". In: *Proceedings of the National Academy of Sciences USA*. 2010.

[39] Jorge Gonzalez-Dominguez, Yongchao Liu, and Bertil Schmidt. "Parallel and Scalable Short-Read Alignment on Multi-Core Clusters Using UPC++". In: *PloS one* 11.1 (2016), e0145490.

[40] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel computing* 22.6 (1996), pp. 789–828.

[41] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. "QUAST: quality assessment tool for genome assemblies". In: *Bioinformatics* 29.8 (2013), pp. 1072–1075.

[42] Bahlul Haider, Tae-Hyuk Ahn, Brian Bushnell, Juanjuan Chai, Alex Copeland, and Chongle Pan. "Omega: an Overlap-graph de novo Assembler for Metagenomics". In: *Bioinformatics* (2014), btu395.

[43] Ayat Hatem, Doruk Bozdağ, Amanda E Toland, and Ümit V Çatalyürek. "Benchmarking short sequence mapping tools". In: *BMC bioinformatics* 14.1 (2013), p. 184.

[44] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[45] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. "Hopscotch hashing". In: *International Symposium on Distributed Computing*. Springer. 2008, pp. 350–364.

[46] Manuel Holtgrewe, Anne-Katrin Emde, David Weese, and Knut Reinert. "A novel and well-defined benchmarking method for second generation read mapping". In: *BMC bioinformatics* 12.1 (2011), p. 210.

[47] Meichun Hsu and Wei-Pang Yang. "Concurrent Operations in Extendible Hashing." In: *VLDB*. Vol. 86. 1986, pp. 25–28.

[48] LA Hug, BJ Baker, K Anantharaman, CT Brown, AJ Probst, and et al. "A new view of the tree of life". In: *Nature Microbiology* (16048 2016).

[49] Martin Hunt, Taisei Kikuchi, Mandy Sanders, Chris Newbold, Matthew Berriman, Thomas D Otto, et al. "REAPR: a universal tool for genome assembly evaluation". In: *Genome Biol* 14.5 (2013), R47.

[50] Parry Husbands, Costin Iancu, and Katherine Yelick. "A Performance Analysis of the Berkeley UPC Compiler". In: *Proc. of International Conference on Supercomputing.* ICS '03. San Francisco, CA, USA: ACM, 2003, pp. 63–73. ISBN: 1-58113-733-8. DOI: `10.1145/782814.782825`.

[51] Benjamin G Jackson, Matthew Regennitter, Xiao Yang, Patrick S Schnable, and Srinivas Aluru. "Parallel de novo assembly of large genomes from high-throughput short reads". In: *IPDPS'10*. IEEE. 2010.

[52] Benjamin G Jackson, Matthew Regennitter, et al. "Parallel de novo assembly of large genomes from high-throughput short reads". In: *IPDPS'10*. IEEE. 2010.

[53] David Jaffe. *Discovar: Assemble genomes and find variants.* `http://www.broadinstitute.org/software/discovar/blog/`. 2014.

[54] *Joint Genome Institute.* `http://jgi.doe.gov`. Accessed: 2016-07-18.

[55] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. "A simple algorithm for finding frequent elements in streams and bags". In: *ACM Transactions on Database Systems (TODS)* 28.1 (2003), pp. 51–55.

[56] John D Kececioglu and Eugene W Myers. "Combinatorial algorithms for DNA sequence assembly". In: *Algorithmica* 13.1-2 (1995), pp. 7–51.

[57] Sergey Koren, Todd J Treangen, and Mihai Pop. "Bambus 2: scaffolding metagenomes". In: *Bioinformatics* 27.21 (2011), pp. 2964–2971.

[58] Sameer Kumar, Amith R Mamidala, Daniel A Faraj, Brian Smith, Michael Blocksome, Bob Cernohous, Douglas Miller, Jeff Parker, Joseph Ratterman, Philip Heidelberger, et al. "PAMI: A parallel active message interface for the Blue Gene/Q supercomputer". In: *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International.* IEEE. 2012, pp. 763–773.

[59] Vijay Kumar. "Concurrent operations on extendible hashing and its performance". In: *Communications of the ACM* 33.6 (1990), pp. 681–694.

[60] Ben Langmead and Steven L Salzberg. "Fast gapped-read alignment with Bowtie 2". In: *Nature methods* 9.4 (2012), pp. 357–359.

[61] Jonathan Laserson, Vladimir Jojic, and Daphne Koller. "Genovo: de novo assembly for metagenomes". In: *Journal of Computational Biology* 18.3 (2011), pp. 429–443.

[62] Dinghua Li, Ruibang Luo, Chi-Man Liu, Chi-Ming Leung, Hing-Fung Ting, Kunihiko Sadakane, Hiroshi Yamashita, and Tak-Wah Lam. "MEGAHIT v1. 0: A fast and scalable metagenome assembler driven by advanced methodologies and community practices". In: *Methods* 102 (2016), pp. 3–11.

[63] Heng Li. "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM". In: *arXiv preprint arXiv:1303.3997* (2013).

[64] Heng Li and Richard Durbin. "Fast and accurate short read alignment with Burrows–Wheeler transform". In: *Bioinformatics* 25.14 (2009), pp. 1754–1760.

[65] Heng Li and Nils Homer. "A survey of sequence alignment algorithms for next-generation sequencing". In: *Briefings in bioinformatics* 11.5 (2010), pp. 473–483.

[66] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. "SOAP: short oligonucleotide alignment program". In: *Bioinformatics* 24.5 (2008), pp. 713–714.

[67] Ruiqiang Li, Hongmei Zhu, et al. "De novo assembly of human genomes with massively parallel short read sequencing". In: *Genome research* 20.2 (2010), pp. 265–272.

[68] Binghang Liu, Yujian Shi, Jianying Yuan, Xuesong Hu, Hao Zhang, Nan Li, Zhenyu Li, Yanxiang Chen, Desheng Mu, and Wei Fan. "Estimation of genomic characteristics by analyzing k-mer frequency in de novo genome projects". In: *arXiv preprint arXiv:1308.2012* (2013).

[69] Yongchao Liu, Bernt Popp, and Bertil Schmidt. "CUSHAW3: sensitive and accurate base-space and color-space short-read alignment with hybrid seeding". In: *PloS one* 9.1 (2014), e86869.

[70] Yongchao Liu and Bertil Schmidt. "CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing". In: *Design & Test, IEEE* 31.1 (2014), pp. 31–39.

[71] Yongchao Liu and Bertil Schmidt. "Long read alignment based on maximal exact match seeds". In: *Bioinformatics* 28.18 (2012), pp. i318–i324.

[72] 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.

[73] Ruibang Luo, Thomas Wong, Jianqiao Zhu, Chi-Man Liu, Xiaoqian Zhu, Edward Wu, Lap-Kei Lee, Haoxiang Lin, Wenjuan Zhu, David W Cheung, et al. "SOAP3-dp: fast, accurate and sensitive GPU-based short read aligner". In: *PloS one* 8.5 (2013), e65632.

[74] Kanak Mahadik, Somali Chaterji, Bowen Zhou, Milind Kulkarni, and Saurabh Bagchi. "Orion: Scaling Genomic Sequence Matching with Fine-Grained Parallelization". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. 2014.

[75] Guillaume Marçais and Carl Kingsford. “A fast, lock-free approach for efficient parallel counting of occurrences of k-mers”. In: *Bioinformatics* 27.6 (2011), pp. 764–770.

[76] Victor M. Markowitz, I-Min A. Chen Chen, Ken Chu, Ernest Szeto, Krishna Palaniappan, Manoj Pillay, Anna Ratner, Jinghua Huang, Ioanna Pagani, Susannah Tringe, Marcel Huntemann, Konstantinos Billis, Neha Varghese, Kristin Tennessen, Konstantinos Mavromatis, Amrita Pati, Natalia N. Ivanova, and Nikos C Kyrpides. “IMG/M 4 version of the integrated metagenome comparative analysis system”. In: *Nucleic Acids Research* 42 (D1 Sept. 2013), pp. D568–D573.

[77] Chris Maynard. “Comparing one-sided communication with MPI, UPC and SHMEM”. In: *Proceedings of the Cray User Group (CUG) 2012* (2012).

[78] 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.

[79] Jintao Meng, Bingqiang Wang, Yanjie Wei, Shengzhong Fen, and Pavan Balaji. “SWAP-Assembler: scalable and efficient genome assembly towards thousands of cores”. In: *Proceedings of the Fourth Annual RECOMB Satellite Workshop on Massively Parallel Sequencing (RECOMB-Seq 2014)*. Pittsburgh, Pennsylvania, USA, 2014.

[80] Rohith K Menon, Goutham P Bhat, and Michael C Schatz. “Rapid parallel genome indexing with MapReduce”. In: *Int. workshop on MapReduce and its applications*. ACM. 2011, pp. 51–58.

[81] Maged M Michael. “High performance dynamic lock-free hash tables and list-based sets”. In: *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM. 2002, pp. 73–82.

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

[83] Sanchit Misra, Ramanathan Narayanan, Wei-keng Liao, Alok Choudhary, and Simon Lin. “pFANGS: Parallel high speed sequence mapping for next generation 454-roche sequencing reads”. In: *IPDPSW*. IEEE. 2010, pp. 1–8.

[84] Toshiaki Namiki, Tsuyoshi Hachiya, Hideaki Tanaka, and Yasubumi Sakakibara. “MetaVelvet: an extension of Velvet assembler to de novo metagenome assembly from short sequence reads”. In: *Nucleic acids research* 40.20 (2012), e155–e155.

[85] Sergey Nurk, Dmitry Meleshko, Anton Korobeynikov, and Pavel Pevzner. “metaSPAdes: a new versatile de novo metagenomics assembler”. In: *arXiv preprint arXiv:1604.03071* (2016).

[86] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. “IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth”. In: *Bioinformatics* 28.11 (2012), pp. 1420–1428.

[87] D Peters, X Luo, K Qiu, and P Liang. "Speeding up large-scale next generation sequencing data analysis with pBWA". In: *J Appl Bioinform Comput Biol* 1 (2012), p. 2.

[88] *pMap: Parallel Sequence Mapping Tool.* `http://bmi.osu.edu/hpc/software/pmap/pmap.html`.

[89] Martin Raab and Angelika Steger. "Balls into Bins: A Simple and Tight Analysis". In: *Randomization and Approximation Techniques in Computer Science*. Springer, 1998, pp. 159–170.

[90] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network*. Vol. 31. 4. ACM, 2001.

[91] C Rinke, P Schwientek, A Sczyrba, NN Ivanova, IJ Anderson, JF Cheng, A Darling, S Malfatti, BK Swan, EA Gies, JA Dodsworth, BP Hedlund, G Tsiamis, SM Sievert, WT Liu, JA Eisen, SJ Hallam, NC Kyrpides, R Stepanauskas, EM Rubin, P Hugenholtz, and T Woyke. "Insights into the phylogeny and coding potential of microbial dark matter". In: *Nature* 499 (7459 July 25, 2013), pp. 431–7.

[92] David P Rodgers. "Improvements in multiprocessor system design". In: *ACM SIGARCH Computer Architecture News*. Vol. 13. 3. IEEE Computer Society Press. 1985, pp. 225–231.

[93] Matthew Ruffalo, Thomas LaFramboise, and Mehmet Koyutürk. "Comparative analysis of algorithms for next-generation sequencing read alignment". In: *Bioinformatics* 27.20 (2011), pp. 2790–2796.

[94] Sophie Schbath, Véronique Martin, Matthias Zytnicki, Julien Fayolle, Valentin Loux, and Jean-François Gibrat. "Mapping reads on a genomic sequence: an algorithmic overview and a practical comparative analysis". In: *Journal of Computational Biology* 19.6 (2012), pp. 796–813.

[95] Ori Shalev and Nir Shavit. "Split-ordered lists: Lock-free extensible hash tables". In: *Journal of the ACM (JACM)* 53.3 (2006), pp. 379–405.

[96] Julian Shun and Guy E Blelloch. "Phase-concurrent hash tables for determinism". In: *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM. 2014, pp. 96–107.

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

[98] JT Simpson and M Pop. "The Theory and Practice of Genome Sequence Assembly". In: *Annu Rev Genomics Hum Genet.* (16 2015), pp. 153–72.

[99] Temple F Smith and Michael S Waterman. "Identification of common molecular subsequences". In: *Journal of molecular biology* 147.1 (1981), pp. 195–197.

[100] *SRA database growth*. `http://www.ncbi.nlm.nih.gov/sra/docs/sragrowth/`. Accessed: 2016-07-18.

[101] *SRA database website.* `http://www.ncbi.nlm.nih.gov/sra/`. Accessed: 2016-08-01.

[102] KA Wetterstrand. “DNA sequencing costs: data from the NHGRI Genome Sequencing Program (GSP)”. In: *National Human Genome Research Institute* (2013).

[103] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[104] T Woyke and EM Rubin. “Evolution. Searching for new branches on the tree of life”. In: *Science* 346 (6210 Nov. 7, 2014), pp. 698–9.

[105] Thomas D Wu and Serban Nacu. “Fast and SNP-tolerant detection of complex variants and splicing in short reads”. In: *Bioinformatics* 26.7 (2010), pp. 873–881.

[106] Matei Zaharia, William J Bolosky, Kristal Curtis, Armando Fox, David Patterson, Scott Shenker, Ion Stoica, Richard M Karp, and Taylor Sittler. “Faster and more accurate sequence alignment with SNAP”. In: *arXiv preprint arXiv:1111.5572* (2011).

[107] Hao Zhang, Yonggang Wen, Haiyong Xie, and Nenghai Yu. *A Survey on Distributed Hash Table (DHT): Theory, Platforms, and Applications.* 2013.

[108] Mengyao Zhao, Wan-Ping Lee, Erik P Garrison, and Gabor T Marth. “SSW Library: An SIMD Smith-Waterman C/C++ Library for Use in Genomic Applications”. In: *PloS one* 8.12 (2013), e82138.

[109] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. “UPC++: a PGAS Extension for C++”. In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International.* IEEE. 2014, pp. 1105–1114.

[110] Aleksey Zimin, Kristian A. Stevens, Marc W. Crepeau, Anne Holtz-Morris, Maxim Koriabine, Guillaume Marçais, Daniela Puiu, Michael Roberts, Jill L. Wergrzyn, Pieter J. de Jong, David B. Neale, Steven L. Salzbert, James A. Yorke, and Charles H. Langley. “Sequencing and Assembly of the 22-Gb Loblolly Pine Genome”. In: *Genetics* 196.3 (Mar. 2014), pp. 875–90.