Parallel Algorithms for *De Novo* Long Read Genome Assembly via Sparse Linear Algebra

by

Giulia Guidi

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Katherine Yelick, Co-chair
Adjunct Assistant Professor Aydın Buluç, Co-chair
Professor James Demmel
Professor Daniel Rokhsar

Summer 2022

Parallel Algorithms for *De Novo* Long Read Genome Assembly via Sparse Linear Algebra

Abstract

Parallel Algorithms for *De Novo* Long Read Genome Assembly via Sparse Linear Algebra

by

Giulia Guidi

Doctor of Philosophy in Computer Sciences

University of California, Berkeley

Professor Katherine Yelick, Co-chair

Adjunct Assistant Professor Aydın Buluç, Co-chair

Significant advances in genome sequencing over the past decade have produced a flood of genomic data that pose enormous computational challenges and require new bioinformatics approaches. As the cost of sequencing has decreased and genomics has become an increasingly important tool for health and the environment, genomic data has grown exponentially, often requiring parallel computing on high-performance computing (HPC) systems. However, genomic applications are often characterized by irregular and unstructured computation and data layout, making them a troublesome target for distributed memory parallelism.

In this dissertation, we show that it is possible to productively write highly parallel code for irregular genomic computation using the appropriate abstraction. Genomic algorithms are often based on graph analysis and processing. For individual graph algorithms, it has been previously shown that graphs can be viewed as sparse matrices and the computations become a series of matrix operations. Here, we take this idea to a new level by demonstrating its applicability and challenges for a data- and computationally-intensive end-to-end application in genomics: *de novo* long-read genome assembly, in which an unknown genome is reconstructed from short, redundant, and erroneous DNA sequences. Our main contribution is the design and development of a set of scalable distributed and parallel algorithms for *de novo* long-read genome assembly that can run on hundreds of nodes of an HPC system, reducing the runtime for mammalian genomes from days on a single processor to less than 20 minutes on a supercomputer. Our algorithms are presented as the Extreme-Scale Long-Read Berkeley Assembler (ELBA) pipeline, which encompasses the major phases of the overlap-layout-consensus paradigm that is most popular for long-read sequencing data. In ELBA, we view assembly through the lens of sparse linear algebra, where the core data structure is a sparse matrix. This dissertation paves the way for a highly productive paradigm for writing massively parallel codes for irregular and unstructured real-world computation.

ELBA is built for HPC systems with high-speed network and batch scheduling. However, we recognize that not every research community has access to government or institutional supercomputing facilities that have the necessary scale (e.g., hundreds of nodes) and hardware characteristics (e.g., a low-latency network) to realize the full potential of massively parallel algorithms such as those we have developed in this work. Thus, we believe that a long-term goal of HPC research is to democratize large-scale computing for science, not only through highly productive programming, but also through widely accessible large-scale resources and systems. As a first step in demonstrating the applicability of the ideas presented in this dissertation to a cloud computing environment, we perform a benchmarking exercise to compare HPC and cloud systems. Our study shows that today's cloud systems can compete with traditional HPC systems, at least at moderate scales, due to significant advances in networking technologies.

To Sieger,

"Once you have had a wonderful dog, a life without one, is a life diminished."

Contents

Acknowledgments

If I were to go back to when I applied to graduate school, I would make the exact same decision to pursue a Ph.D. at UC Berkeley under the supervision of Aydın Buluç and Kathy Yelick. I cannot thank Aydın and Kathy enough for their constant support and guidance throughout my PhD.

I first met Aydın in 2017 during a summer internship at the Lawrence Berkeley National Laboratory (LBNL). Aydın's passion and willingness to teach have undeniably influenced my career. If I had not met him, I probably would not have applied to graduate school in the United States. The research experience as an intern at LBNL was incredible and when Aydın offered to write a letter of recommendation, I decided to apply to graduate school in the United States, and the rest is history. Aydın's expertise in parallel programming and sparse matrix computation was crucial to the development of my dissertation and research vision. I'm grateful not only for the technical support he provided me during my PhD, but also for his frequent advice on how to write a paper and present my work. I believe that these soft competencies were almost as important as the technical competencies in the development of my research, allowing me to effectively promote my work and thus make important connection with colleagues both inside and outside of UC Berkeley and LBNL. In these unprecedented times (an overused phrase at this point), a special mention goes to the effort that Aydın put to ensure that his advisees did not feel rattled in the early stages of the pandemic, ranging from a (virtual) "tea hour" simply chatting and checking in with each other rather than discussing technical work, to (masked) outdoor hikes.

Kathy's expertise in parallel computing, programming languages, and domain science has been critical to the development of my dissertation and research vision, and has highlighted its importance and impact on the broader parallel computing research community. Kathy has an undeniable superpower: no matter how busy she is or how many duties she has to handle at once, she always asks the right question or gives the right feedback (even if sometimes you do not know it at the time). In retrospect, thinking back on our conversations over the years, I feel that she has often been able to anticipate challenges or outcomes that would not have been clear to me until days, weeks or months later. I distinctly remember saying, "Oh, Kathy was right!" or "Oh, that's what she meant!" several times over the years. One could argue that it was because I did not always listen to her (which is probably partly true), but a big part of it has to do with the fact that Kathy is just very resourceful and incredibly good at her job. Kathy's advice also went far beyond purely technical circumstances and led to insightful career advice that was especially important when I applied for a faculty position last year. I would like to think I returned the "advising favor" a bit (a hundredth of what she gave me) when I gave her (unsolicited, of course) advice on puppies.

I feel incredibly fortunate to have had the opportunity to work with and learn from them. Kathy and Aydın, also known as Kaydin (to me and Alok), are undoubtedly two people who would answer the question, "Who do you want to be when you grow up?". I look forward to continuing to work with and learn from them as I embark on the next chapter of my

academic career. I would not be the researcher I am today without their mentorship and I hope to be as good an advisor to my advisees as Aydın and Kathy were to me.

I also want to thank Jim Demmel and Dan Rokhsar for agreeing to serve on this dissertation committee and for providing me with useful feedback, but most of all, I want to thank both of them for being a mentor throughout my PhD. I met (more or less) weekly with Jim for the BeBOP group meeting and had the opportunity to pick his mind on a variety of technical (and other) challenges. Jim also provided invaluable support when I applied for a faculty position and helped me through the offer negotiation phase. I have worked with Dan since the beginning of my PhD. His expertise in molecular biology, and genome assembly in particular, has been instrumental in ensuring that the product of this dissertation, ELBA, is not only highly parallel but also biologically sound. Dan also supported me during my job application and provided feedback. I would also like to thank Joe Hellerstein, who was on my qualifying committee and provided helpful feedback for the advancement of this dissertation, and Rob Bisseling of Utrecht University, who, although not formally involved in my dissertation, offered his help and kind opinion, which I especially appreciated during the job application process. I also want to thank Natacha Crooks, James Larus, Olaf Schenk, David Culler, Teodoro Laino, Laura Grigori, Richard Vuduc, and Sunita Chandrasekaran for the advice and support they gave throughout different phases of my PhD.

This dissertation would not be the same without the help I received from coauthors and colleagues throughout my PhD. I would also have had much less fun without them. In particular, I would like to thank Marquita Ellis, my academic sister, who was an invaluable support to me at the beginning of my PhD and with whom I collaborated extensively. In the second half of my PhD, I had the opportunity to work with Oguz Selvitopi, from whom I learned a lot and from whom I will gladly continue to learn. Over the past year, I have had the opportunity to work with and mentor Gabe Raulet. The results achieved last year would not have been possible without his incredible work. I look forward to watching Gabe grow as a researcher. I also want to thank the entire Exabiome Team at LBNL for the incredible growth opportunities each member has provided me. I would especially like to thank Lenny Oliker, who has always been very helpful and supportive (sorry for all the paperwork you had to do for conference approval!).

The truth is that I would never have made it through graduate school without the help and support of the graduate student advisor Shirley Salanio and Jean Nguyen. They were constantly on top of everything, promptly answered any doubt that came to my mind. I tried to repay them by sending them weekly pictures of puppies.

Graduate school was an incredible academic experience, but I owe much of my happiness during those years to my friends who lifted me up when I was sad and shared and celebrated the victories with me. I'm lucky to have an incredible number of great people in my life, too many to thank here, but a special mention goes to Alok Tripathy, who has been my PhD buddy since joining our group in 2019. I cannot imagine going through graduate school without the kind of friend I found in him. I wish everyone had a PhD buddy like Alok in graduate school and life in general (why one would need to suffer through a PhD to deserve a great friend). Chiara Motta has also been a pillar during my PhD. Chiara and I are from

the same hometown in Northen Italy and the universe brought us back together in Berkeley to pursue our PhDs. Chiara is one of the nicest people I have ever met, and yet she was one of the few who had the courage to plainly put me in my place when I needed it most, and for that I'm grateful (for context, arguing with me is not pleasant, especially when you are such a nice person as Chiara). I'm grateful for the friendship I have been able to build with Mae Milano over the past year. Mae was a rock during the job application season, and her enthusiasm for Cornell (where she earned her PhD) was instrumental in my accepting Cornell's offer. I hope that one day (soon) she will join me as a colleague.

In the category of "creatures I love but who are not family or my dogs (who have their own paragraph)" I want also mention my partner, Eli Kinigstein. I met Eli late in my PhD, but his love and support have helped me get through the last few miles. Most of all, his calming presence and constant support have made the dreading feeling (being a little over dramatic) of writing a dissertation much more bearable. It has also helped to greet me with a glass of wine in the evening. I look forward to (our vacation and) continuing to share the passion we both have for our work, supporting each other, and growing together.

A big thank you goes out to Nathan Lambert and Erin Grant, with whom I co-founded and co-organized the Equal Access to Application Assistance (EAAA) program to provide feedback to underrepresented minorities during graduate school application season. It is incredible to know that our effort will live on.

I also want to thank Nikita Samarin, Savannah Gupton, Lars Tatum, Nurbek Tazhimbetov, Sara Mirandola, Valeria Luraghi, Juliane Reinhardt, Neesha Zerin (and baby Abraham), Grace Dihn, Eleonora Bianchi, Emanuele Lagazzi, Eleonora Losiouk, Piergiuseppe Mallozzi, Giuseppe Franco, Andrea Fanelli, Utkarsha Agwan and Akshit Tyagi, Giorgio Berardini, Can Firtina, Lorenzo Di Tucci, Alberto Zeni, Rui Oliveira, Gaia Andreoletti, Vivek Bharadwaj, Gabriel Colon-Reyes, Benjamin Brock, Elizabeth Koning, Jan Willem-Buurlage, Thibault Cimic, Ed Younis, Brock Hudson, Philip and Catrina Mallon, Aariz Hazari, Yocelyn Gutierrez Guerrero, and Federico Dallo. Thank you to all the Twitter buddies I have met along the way. I also thank the amazing people I have met on the job market with whom I have shared the struggle of the past year. I want especially to thank Ethan Cecchetti, Sara Beery, Rachee Singh (and Pistachios), and Grant Ho.

If you know me, you know how much I love and have loved my dogs. I adopted Sieger, a senior black and silver long haired German Shepherd, in 2019. He accompanied me through graduate school as in the official role of PhDog before passing away this spring when he was sure I would graduate. I cannot express how much he meant to me and I miss him every day. This dissertation is for him. This spring I adopted Mahler, a bi-colored German Shepherd puppy who kept me company for the last few miles while I wrote my dissertation. He drives me crazy with his energy (and will continue to do so for a while), but that's on purpose because he didn't want me to be bored while I wrote all day. I look forward to watching him grow and sharing the next chapter of my life with him (I hope he likes the snow). I once met someone who, when I told her what kind of dog I had, said, "Oh yes, you are definitely a German Shepherd person.". That is still the best compliment I have received so far.

Finally, a sincere thank you goes to my family as their constant support never made me feel alone even if have more than 9,000 miles separating us. I'm forever grateful to my mom, Paola, and dad, Gian Luca, for giving me the freedom to do what I thought was best, even if it was hard for them to let go. This freedom allowed me to be the person I am today, to bang my head against a wall and learn, to share what I learned with them and grow together as a family. I miss them and my sister, Gaia (the funniest person on the planet), every day and I hope that one day I'll be able to combine my career with being closer to them. I thought I was getting closer to them when I took a job at Cornell in upstate New York, but it turns out that a flight from Ithaca to Milan takes as long as a flight from San Francisco to Milan. Oh well, they'll come to visit me in Ithaca because the housing market is much nicer there and I can afford a decent-sized house to host them, and a hot tub for Alok (inside joke).

Thank you for making my PhD, and my life, so beautiful.

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

Chapter 1

Introduction

In the last decade, we have seen significant advances in genome sequencing technologies. These technologies enable the extraction of nucleotide (i.e., the basic structural unit of genomes) sequences from biological samples using chemical and physical processes. As the cost of sequencing declines and genomics permeates various scenarios of our lives, the need for scalable computing and data systems becomes increasingly important—often fundamental—to drive scientific discovery and bring such advances into the real world [162]. **As a flood of data is generated, applications in genomics require the full computational resources of institutional high-performance computing (HPC) systems.**

This dissertation demonstrates how it is possible to productively write parallel code for irregular and unstructured problems using the appropriate abstraction. A major contribution of this dissertation is the design and development of a set of scalable distributed and parallel algorithms for genome analysis that can run on hundreds of nodes of an HPC system using the powerful sparse matrix abstraction. GraphBLAS [98, 47] has shown that sparse matrices are a good abstraction for irregular graph algorithms. This dissertation takes the sparse matrix abstraction to a new level by demonstrating its applicability and challenges for a real end-to-end application: the problem of *de novo* long read genome assembly. This dissertation opens the door to high-performance genomics and addresses some of the computational challenges that limit the far-reaching impact of genomics on daily life.

The reality, however, is that not every research community has easy access to government or institutional supercomputing facilities. The problem is exacerbated for communication-intensive and irregular applications—common in computational genomics—that require scalable hardware and a low-latency network. HPC systems are typically allocated to specific research communities and have long user wait times, limiting access to their resources and thus scientific discoveries. Therefore, **it is not enough to provide a user with a scalable algorithm because they may not have access to large-scale resources.** The long-term goal is to democratize large-scale computing and make it more accessible to the scientific community. The second key contribution of this dissertation is a performance study of today's cloud computing systems, showing that cloud computing can now compete with traditional HPC systems, at least at moderate scales, and initiate a paradigm shift in high-performance computing for scientific computing.

1.1 Challenges of Parallelization

Genomic applications typically use shared-memory parallelism, which is the kind of parallelism we can have on a single processor when the work is divided among multiple threads or processes (e.g., one process per core on a 32-core processor) and they have access to a common (shared) memory. They typically implement this multithreaded parallelism using the POSIX Threads [106] programming model, commonly known as pthreads, or the OpenMP [129] programming model, and the best performing codes often also use data parallelism through the SIMD (Single Instruction Multiple Data) [63] instruction set in conjunction with multithreading. This programming model is suitable for irregular and unstructured applications because they require fine-grained communication between processes. This model provides much faster access to memory than distributed memory parallelization, which requires communication over the network, and perhaps most importantly, it greatly simplifies the programmer's considerations for parallelizing the application.

Despite the productivity advantage, shared memory parallelism alone is not adequate to handle the massive volumes of data currently being generated in genomics, as algorithm scaling and absolute performance is limited by the number of cores on a single processor. Distributed memory parallelism is a natural solution to scaling and performance issues, but it comes at the price of lower productivity and a steep learning curve for a novice programmer, since communication between processes must be handled explicitly, such as in the Message Passing Interface (MPI) programming model [82], where if a process P_i requires a piece of data that P_j stores in its local memory, P_j must explicitly send that information through a message over the network to P_i, which must be ready to receive it by setting up a receive buffer. This type of communication is called synchronous because the process receiving the data cannot proceed with its local execution until the data transfer is complete. It is straightforward to understand that this programming model makes writing parallel code for irregular and unstructured problems a significant productivity burden and performance barrier (since communication is more expensive than computation), since we have to send many small messages (fine-grained communication) over the network and wait until the transfer is complete before proceeding with execution.

Parallel computing research comes to the rescue with a programming model paradigm called Partitioned Global Address Space (PGAS) [4, 72, 37], whose goal is to combine the best of shared memory parallelism with the best of distributed memory parallelism. PGAS defines a global memory address space abstraction that is logically partitioned, where a portion is local to each process. Thus, PGAS is a distributed memory programming model paradigm that gives the programmer the semblance of writing a program for shared memory parallelism. A programming model that uses the PGAS paradigm, such as UPC++ [12], often gives the programmer the ability to use asynchronous or one-sided communication, such as through Remote Memory Access (RMA), where one process can directly access memory with affinity to another (potentially remote) process without the explicit semantic involvement of the passive target process. That is, in a PGAS model P_j would not have to explicitly send the data to P_i in our previous example, but P_i could retrieve this information

directly with an RMA operation. The one-sided communication aspect makes a PGAS-based programming model a suitable choice for implementing irregular and unstructured computations in distributed memory [70, 93, 20]. However, the performance and scaling of algorithms implemented using the PGAS paradigm are limited by the latency of the network offered by supercomputing facilities, since PGAS often involves many frequent single accesses to distributed data structures.

In this dissertation, we show that there is another way to implement parallel algorithms for irregular and unstructured computations in distributed memory. On thousands of cores, we demonstrate high performance and near-linear scaling without sacrificing productivity and without frequent individual remote accesses. The key idea to simultaneously achieve high performance and productivity with an bulk synchronous approach is to **map irregular computation to computational primitives that are better suited for distributed-memory parallelism**. Here, we first consider that graph structures and graph algorithms are widely used in genomics and more generally in biology, since it is common in this field to search for relationships and similarities within and between organisms. Then, we consider that a graph is nothing more than a sparse matrix. Therefore, rather than using hash tables to represent the graph, we use sparse matrices as the core data structures. The computation is then modeled as computation between sparse matrices, e.g., sparse matrix multiplication and sparse matrix-vector multiplication. GraphBLAS [98, 47] has proved that sparse matrices are a suitable abstraction for graph algorithms. In this dissertation, we show how it is possible to exploit the duality between a graph and a sparse matrix to productively write high-performance code for a real-world end-to-end application in genomics, namely *de novo* long read assembly.

A graph can be seen as a sparse matrix, but often a genomic graph is not concerned with purely numerical values. Consequently, matrix multiplication, where we try to sum and multiply nucleotide sequences, for example, would not be useful for the application. A key element to the success of this approach is the algebraic concept of semiring abstraction, which gives us the ability to overload the addition and multiplication operator of matrix multiplication with any operator that makes sense for the application. For example, the Floyd-Warshall shortest path algorithm can be reformulated as a computation over a $(\min, +)$ semiring.

The combination of sparse matrix abstraction and semiring abstraction provides an extremely powerful way to implement irregular and unstructured computation. On the one hand, sparse matrix abstraction allows us to implement highly parallel codes in distributed memory without sacrificing productivity because we in the HPC research community know how to efficiently implement such computation and in general, once we are able to map a computation to a primitive, we can continuously and seemingly effortlessly benefit from new advances in that primitive. On the other hand, semiring abstraction provides us with the flexibility and modularity that is so important in genomics (and in a scientific application in general), as we want to be able to accurately model and process the data without being constrained in how we process it to avoid performance degradation.

Overall, we believe that the parallelization challenges we address in this dissertation have implications for scientific computing in general beyond *de novo* long read assembly and ge-

nomics, as many irregular and unstructured algorithms currently rely on hash tables and graph structures. The ability to productively write high-performance code is a major milestone in democratizing parallel computing for science to enable faster, high-quality scientific discovery, and it will become increasingly important as large-scale parallel resources become more necessary.

Democratizing large-scale computing and making it more accessible to the genomics community and the scientific community at large requires both widespread access to resources and knowledge of how to use resources efficiently to develop scalable algorithms. The latter without the former is only part of the solution. As the second main contribution of this dissertation, we investigate the performance of on-premise and cloud-based HPC systems and show that today's cloud for scientific computing can compete with traditional HPC systems, at least at moderate scales. Cloud computing has made significant advances in networking technology and HPC system software that have overturned the state of the art and opened the door for a paradigm shift in high-performance computing.

Until 2018, the cloud was not an option for much scientific software due to the lack of a low-latency network [125]. In this dissertation, we investigate the performance gap between traditional and cloud-based HPC systems to understand the nature of their differences and guide the design of future high-performance systems. To this end, we analyze the cross-stack performance, from single core compute power, to memory subsystem, inter-node communication performance, and overall application performance.

Our results show that cloud platforms with similar processors and networks can achieve HPC-competitive performance not only for compute-intensive applications, but also for communication-intensive applications. Today's cloud computing has overcome one of its main limitations—at least at small and medium scales—by providing higher-speed memory and interconnects for HPC-oriented instances.

In addition, we highlight that cloud computing can provide a greater variety of hardware configurations and newer technologies due to continuous procurement cycles. If a study requires the latest technology or a specific memory size and processor type, these are more likely to be available in the cloud, whereas a given HPC system may offer only one or a small set of standardized resources suitable for typical scientific applications.

1.2 Overview of ELBA, a Long-Read Assembler

Despite the advances that sequencing companies have made in recent years, these technologies can only read and output nucleotide sequences in the form of a string of limited length. This length ranges from 100 to 250 nucleotides for short-read technologies and from 5,000 to 100,000 for long-read technologies. The sequences generated by the sequencing process are referred to as *reads*. In both cases, the read length is much smaller than the size of the genome, e.g., the human genome consists of 3 billion nucleotides, while the wheat genome has 17 billion nucleotides. In terms of cost, short-read technologies are currently cheaper per nucleotide sequenced than long-read technologies.

Historically, the longer read length of long-read technologies has come at the cost of higher error rates, averaging 15-35%, as opposed to 0.1% for short-read technologies, where an error is the insertion, deletion, or substitution of a nucleotide in the read. However, the advantages derived from improved length have led to the increasing popularity of long-read technologies either as a stand-alone tool or in combination with short-read technologies. To understand one of the key advantages of long read sequencing technologies, we need to keep in mind that the genomic sequence that makes up our DNA is not completely unique, as genomes contain repeated sequences (also known as repeats), which are patterns of nucleotides that occur in multiple copies throughout the genome.

These repeated sequences pose a major challenge for a critical and computationally intensive application that we also focus on in this dissertation, namely *de novo* genome assembly. The *de novo* assembly process consists of recreating an unknown genome from the redundant, erroneous, and fragmentary genomic reads obtained from sequencing machines. Longer sequences make it easier to resolve the repeat, that is, to know the region of the genome from which a particular repeat copy originated, because the length of the read is often longer than the repeat itself, so we can correctly reconstruct the sequence around a repeat using the non-repeat portion of the read. Because the sequences in short-read sequencing are often shorter than the repeat region, it is much more challenging to resolve the repeat [14]. The longer the read, the better it is for resolving the repeat. In 2019, Pacific Biosciences unveiled a new long-read technology called HiFi that generates sequences between 5,000 and 10,000 nucleotides with error rates around or below 0.5% [158], increasing the popularity and adoption of long-read technologies. For this reason, this dissertation focuses on long-read sequencing technologies.

De novo genome assembly is often a necessary step in bioinformatics, e.g., for comparative genomics analyses and for conservation genetics applications [152] as well as for studies of genomic variation in various diseases, such as Alzheimer's disease [126]. It is both data and computationally intensive, as the input can be as large as terabytes and the computation involves expensive procedures. Reconstruction of an unknown mammalian genome can take days on a 48-core processor, making extensive studies impractical or infeasible. For plant genomes, which can be up to two orders of magnitude larger than mammalian genomes, the memory requirement and the runtime are even higher.

Parallel implementation of the assembly process on a supercomputer can reduce runtime and enable faster assembly of larger genomes. Yet, **scalable parallel programming in distributed-memory is difficult and not popular in genomics**.

This dissertation addresses the computational challenges of assembling long-read sequencing data and proposes the Extreme-Scale Long-Read Berkeley Assembler (ELBA) [83, 85, 84]. Assembling large genomes, such as plant genomes, is currently impractical or infeasible due to memory and computational requirements. However, even with mammalian genomes, which are large but generally smaller than plant genomes, the lack of scalability is a limitation because assembly is often performed multiple times within a given study. Researchers may want to optimize the algorithm settings for a particular genome or input, or they may want to assemble and compare multiple entities of the same species, as is the case in pange-

nomics, where the genome is represented as a graph, with identical regions between entities represented as a linear sequence, while differences take the form of a graph [149]. Considering that *de novo* assembly of a genome is rarely a one-time experiment, it can also become impractical for mammalian genomes, as each assembly can take many hours or days.

ELBA is the first long-read assembler implemented for distributed-memory parallelism, and it introduces the use of sparse matrices as main data structures. Using matrices enables the development of new algorithms based on linear algebra that facilitate parallelization across hundreds of nodes and improve computational modularity and portability. **ELBA enables the runtime for mammalian genomes to be reduced from days to less than 30 minutes on a supercomputer**, opening the door to assembling unknown large genomes that were previously impractical.

In *de novo* genome assembly, there are two main strategies to follow when assembling genomes: De Bruijn graph-based paradigm and Overlap-Layout-Consensus (OLC) paradigm. The choice of one over the other is mainly determined by the sequencing technologies targeted by the algorithm. During assembly using a De Bruijn graph, a directed graph representing overlapping matches between nucleotide sequences, reads are divided into smaller subsequences of fixed size k, called k-mers. The k-mers are then used as nodes in the graph assembly. The nodes that overlap by a certain amount (usually k-1) are then connected by an edge. The assembly is then done by traversing the De Bruijn graph. This strategy is not suitable for sequences with high error rates, as is often the case with long read sequences, since any error in a read creates a bulge in the De Bruijn graph, making assembly difficult [43]. Bulges are also the case when there are imprecise repeats in a genome (e.g., two regions that differ by a single nucleotide or other small variation). In addition, it is common practice to use the De Bruijn graph approach for short-read data because computing overlap candidates using an OLC strategy for short-read sequences is extremely computationally intensive, since short-read technologies sequence with much higher coverage than long-read technologies and their length is on average 100× shorter than long-read sequences. That would mean 100× more sequence comparisons for the same depth. Therefore, De Bruijn is the usual choice for short-read technologies whose error rate is very low, while the OLC paradigm is the most common assembly strategy for long-read sequencing data [16].

The overlap step (O) consists of identifying overlapping regions between input sequences to create an *overlap* graph. In an overlap graph, the vertices are the input sequences and the edges are the overlap regions between two connected sequences in the graph. Because of redundant sequencing and the inherent genome repetitiveness, the layout step (L) simplifies the overlap graph and converts it into a *string* graph, i.e., a graph where sequences are vertices and edges are the suffixes of an overlap between two sequences (i.e., if v_1 = ATGCC and v_2 = GCCTT are two overlapping sequences, then TT is the suffix between v_1 and v_2), and there is no redundancy with respect to vertices and edges. A redundant vertex is a sequence that completely overlaps with another sequence (i.e., it is contained in another sequence), while a redundant edge is a transitive edge (i.e., there is a better path in the graph that connects the sequences under consideration). In the consensus step (C), the string graph is processed to obtain a first draft of the unknown genome, returning a *contig*

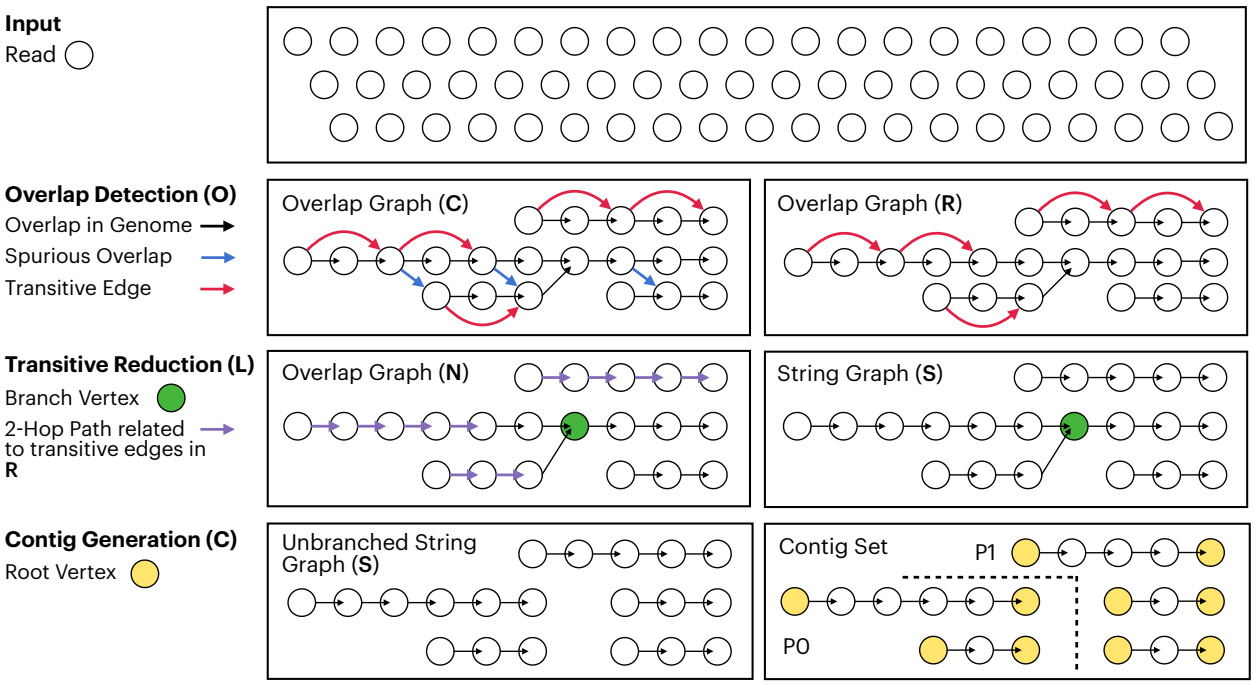

Figure 1.1: A high-level representation of the OLC assembly paradigm as implemented in this dissertation.

set as output. A contig is a group of overlapping DNA sequences that together form a region of a chromosome. Figure 1.1 illustrates the OLC paradigm as implemented by ELBA.

ELBA is developed for long read sequencing, so it follows the OLC paradigm and has five main stages: (i) k-mer counting, where subsequences of fixed length k are extracted from the input sequences and counted to produce a histogram of frequencies; (ii) overlap detection, where the k-mer set is used in conjunction with the input sequences to find initial candidate matches between these sequences; (iii) pairwise alignment, in which false-positive matches are removed; (iv) transitive reduction, in which matches between sequences are considered from a global perspective to remove redundant matches that only complicate the assembly process; and (v) contig generation, in which subsequent linear matches between sequences are identified to create longer contiguous regions of the genome (i.e., contigs).

In ELBA, every step in encoded as a sparse matrix (or multiple matrices). The first stage of ELBA, i.e., k-mer counting, incorporates novel theoretical methodologies, originally presented in our work BELLA [83], to select the optimal value of k and filter out uninformative k-mer subsequences, e.g., repetitive k-mers, to ensure accuracy of the results in addition to efficient parallelization. These methodologies include a general Markov chain model showing that a k-mer seed-based approach is useful to identify sequences that originate from the same genomic region (i.e., overlap candidates) and a method to prune the k-mer set based on the k-mer frequency in the input dataset, the value of k, and the error rate of the dataset. Our

distributed-memory implementation of the k-mer counter is similar to that of HipMer [70] and diBELLA 1D [57], except for the changes required to implement the theoretical methodologies presented in BELLA [83], and is composed of two phases. This implementation uses the Bloom filter, a probabilistic data structure for querying the membership of an element, which serves to avoid storing k-mers that occur only once in the input dataset, since they would lead to no match between sequences and would only waste memory. First, the k-mers are added to the Bloom filter and then the frequencies for the filtered k-mers are calculated. Processes in distributed memory extract k-mers from their local sequences, hash them, and possibly communicate them to other processes as dictated by the Bloom filter's hash function. On the receiving process, the incoming k-mers are added to the local Bloom filter; if they already exist, they are added to the local hash table partition. The communication requires an all-to-all exchange and is implemented using MPI Alltoall and MPI Alltoallv. The output of this phase is a 1D distributed hash table where k-mers instances are the key and frequency and the origin sequences are the values. This hash table can be viewed as a sparse matrix representation, which is a $|k-mers|$-by-$|sequences|$ matrix that we call $\mathbf{A}^\mathsf{T}$.

The second stage of ELBA introduces the sparse matrix abstraction for overlap detection. This innovation was first proposed in our work BELLA [83] in shared memory and then in our work diBELLA 2D [85] in distributed memory. In overlap detection, as well as in the subsequent stages of transitive reduction and contig generation, we map irregular complex algorithms to linear algebra with semiring generalization, which provides automatic scaling and portability. The use of sparse matrices throughout the computation reduces the need for various data structures typically used in genome assembly. Careful design of the semiring is critical to ensure quality and correctness, while careful design of the sparse computation is essential for high performance. Overlap detection takes as input the k-mer hash table and each process on a 2-dimensional distributed processor grid uses the local k-mer hash table and local sequences to create a distributed $|sequences|$-by-$|k\text{-}mers|$ matrix $\mathbf{A}$. $\mathbf{A}$ is multiplied by its transpose $\mathbf{A}^\mathsf{T}$ to obtain the sparse candidate overlap matrix $\mathbf{C}$ of dimension $|sequences|$-by-$|sequences|$. In $\mathbf{C}$, any nonzero $\mathbf{C}_{ij}$ signifies that sequence i and sequence j have at least one k-mer in common. For distributed sparse matrices and computation on them, ELBA relies on the Combinatorial BLAS (CombBLAS) library [29], a parallel distributed-memory graph library that allows working with sparse matrices using a semiring abstraction. CombBLAS relies on the 2D Sparse SUMMA algorithm for parallel SpGEMM [24] and it uses a hybrid hash table and heap based algorithm for local multiplication. This computation is part of the Overlap stage of the assembly paradigm and it is summarized in Figure 1.1.

Once the candidate matrix $\mathbf{C}$ is formed, ELBA computes a round of pairwise alignment, i.e., the process of aligning two sequences so that the similarity areas are maximized. From the matrix point of view, pairwise alignment is an in-place element-wise operation on $\mathbf{C}$ that decides whether the similarity between two sequences is large enough to be considered a true overlap. ELBA then computes another in-place operation on $\mathbf{C}$ that removes nonzeros whose similarity threshold is not above a predefined threshold (e.g., the blue arrow in Figure 1.1). ELBA includes a novel method for discriminating between true and false-

positive candidates based on a Chernoff bound, originally presented in BELLA [83], which is used in deciding the similarity threshold based on the region of overlap between sequences. The resulting matrix $\mathbf{R}$ is the input for the next stage, i.e., the transitive reduction, which is the Layout stage in the assembly paradigm in Figure 1.1.

Our distributed memory transitive reduction algorithm takes the overlap matrix (or overlap graph) $\mathbf{R}$ as input and computes a transitive reduced version of $\mathbf{R}$, which we call the string matrix (or string graph) $\mathbf{S}$. As in the previous phase, we use sparse matrices and computation on them to perform the transitive reduction. A string graph is an overlap graph without redundant vertices and edges. A redundant vertex is defined as a sequence that is completely contained in another sequence and therefore can be removed. Contained sequences are identified and removed from $\mathbf{R}$ before transitive reduction is performed. Due to redundancy in the input sequences, we can have parallel edges in $\mathbf{R}$. If we consider three vertices $\{v_1, v_2, v_3\}$ (i.e., sequences) and edges between any two of these three, we can go from v_1 to v_3 either via v_2 or from v_1 directly to v_3 (e.g., the magenta arrow in Figure 1.1). The weight of the edges gives information about the length of the overlap region between two sequences. Thus, if we have two parallel paths from v_1 to v_3, they either contain the same overlap length or, more likely because the sequences are faulty, one of them stores a longer overlap region. That is, one path is considered more informative. Therefore, we can mark the less informative edge(s) as *transitive*, i.e., redundant, and remove them from the string graph. Our transitive reduction is an iterative procedure in which the matrix $\mathbf{R}$ is repeatedly squared to obtain the n-hop *neighbor* matrix $\mathbf{N}$, which is then compared to $\mathbf{R}$ to see if a parallel n-hop path stores more information than a one-hop path in the original matrix. If so, the original nonzero is marked as transitive and removed from $\mathbf{R}$ as soon as the iterative process is complete, i.e., when the number of transitive edges is unchanged. Let us assume that the purple two-hop path in Figure 1.1 has more information than the magenta path, so we remove the magenta path from $\mathbf{R}$. The output is a string graph $\mathbf{S}$ without redundant edges and vertices. A string graph has the desirable property of collapsing genomic repetitive areas into a single unit [148], which facilitates the following and final task, i.e., contig generation. This algorithm was originally presented in our work diBELLA 2D [85], but was modified during the development of the contig generation step. ELBA's transitive reduction algorithm achieved a 13× speedup for the human genome on more than 300 nodes compared to a distributed memory competitor [133].

As a final step, ELBA introduces a novel distributed memory algorithm that generates the contig set starting from $\mathbf{S}$ and using sparse matrix abstraction for the Consensus stage of the assembly pipeline. The algorithm first determines the contig set starting from $\mathbf{S}$ by masking out the branches (i.e., it creates the unbranched string graph $\mathbf{S}$ in Figure 1.1) and extracting the unbranched paths by computing connected components over the graph. Using a greedy multiway number partitioning algorithm, ELBA then determines how to redistribute sequences (i.e., rows and columns of the sparse matrix) between processes so that sequences belonging to the same contig are stored on the same processor. Once the sequences-to-processor assignment is computed, ELBA uses this information to redistribute the sequences so that each processor has a local sparse matrix representing the unbranched

contig(s) assigned to the processor. ELBA then computes a local assembly step on each processor independently and in parallel, and then returns the contig set. Our contig generation algorithm is fast and efficient because it localizes the contig assembly, so that most of the computation can be performed locally without requiring fine-grained communication. Compared to related work [70], this approach significantly reduces the fraction of runtime required by the entire assembly pipeline for contig generation.

ELBA is the first distributed- memory implementation of *de novo* long-read genome assembly, and it demonstrates good scaling with a parallel efficiency of 80% on 128 nodes on a representative dataset and on two different machines. ELBA achieved speedups of up to 36× and 159× for the O. sativa dataset compared to two state-of-the-art software packages, paving the way for high-performance genome assembly. Our approach results in uniform coverage of the genome and shows promising results in terms of assembly quality, which is measured as completeness (i.e., percentage of the reference genome to which at least one contig was aligned), longest contig size, number of contigs, and misassembled contigs (i.e., number of contigs containing incorrect assemblies, e.g., a contig consisting of sequences derived from different regions of the reference genome).

1.3 Contributions

The main contribution of this dissertation is to demonstrate how we can use sparse matrices in a scientific application to productively write parallel code without sacrificing performance. The core idea is that we can achieve high performance without sacrificing programmer productivity if we can map irregular computations to computational primitives that are better suited for distributed-memory parallelism. The implications of this dissertation go beyond *de novo* long-read assembly and genomics to affect the way we think about irregular and unstructured computations in general. In particular:

- In Chapter 3, we describe a new parallel algorithm for overlap detection using sparse matrices and demonstrate its performance and productivity advantages over a distributed hash table-based implementation.
- In Chapter 4, we present a new parallel algorithm for transitive reduction using sparse matrices and show a speedup of up to 29× over a distributed memory implementation based on Apache Spark.
- In Chapter 5, we introduce a new parallel algorithm for contig generation that localizes computation and minimizes communication by using sparse matrices, and demonstrate the end-to-end performance improvement of up to 159× on the human genome over shared-memory-based state-of-the-art software.

In addition, this dissertation introduces mathematical modeling of long-read sequencing data to achieve not only performance objectives, but also quality objectives essential to the application under study. The following methodologies are described in Chapter 3:

- A new Markov chain-based model that proves that a seed-based approach is suitable for assembling long-read sequencing data and provides us with the optimal seed length based on the error rate of the dataset and sequencing coverage.
- A new seed set pruning method that preserves almost all true overlap matches with high probability, increasing computational efficiency without loss of accuracy.
- A Chernoff bound-based method to separate true overlap candidates from false positives using the similarity score of the alignment between any pair of sequences.

In Chapter 6, we show the potential of cloud-based HPC for data analytics as an alternative to on-premises HPC, because easy access to HPC resources is the other side of the same coin when it comes to democratizing large-scale computing for science:

- A study showing that cloud machines can achieve HPC-competitive performance with processor and network similar to institutional HPC machines, not only for compute-intensive applications but also for communication-intensive applications, overturning the literature and opening the door to a paradigm shift in high-performance computing.

In summary, the remainder of this dissertation is organized as follows. Chapter 2 describes the basic methodologies on which we rely in this dissertation. Chapter 3 describes the foundation, parallel algorithms, and experimental results for overlap detection. Similarly, Chapter 4 and Chapter 5 present the foundation, parallel algorithms, and experimental results for transitive reduction and contig generation, respectively. Chapter 6 describes a performance study on cloud-based HPC systems and illustrates the experimental results compared to traditional HPC systems. Finally, related work is presented in Chapter 7 and Chapter 8 concludes this dissertation.

Chapter 2

Background

In this chapter we describe the basic methodologies on which we base this dissertation. First, we provide an overview of genomics and the long read *de novo*genome assembly problem and its associated paradigm. This paradigm forms the basis for our parallel algorithms. Then, we give an overview of the Combinatorial BLAS library and its distributed sparse matrix computation, which is the core of our parallelization techniques.

2.1 Genome and Genomics

Deoxyribonucleic acid (DNA) is the chemical compound that stores the information needed to develop and control the activities of an organism. DNA molecules consist of two complementary strands arranged in the form of a double helix. One strand defines the *reverse complement* of the other. Each strand is made up of a combination of four molecules called nucleotides or bases, which form the genetic alphabet. Bases are adenine (A), thymine (T), guanine (G), and cytosine (C). Each base has a complementary base on the opposite strand: A is the complement of T, C is the complement of G, and vice versa. A base, together with the corresponding base on the opposite strand, is called a base pair (bp). If v = ATTCG, its reserve complement is v' = CGAAT. The *canonical form* of a DNA sequence v is the lexicographically smaller of v and its reverse-complement v'. In our example, v = ATTCG is the canonical form. The complete DNA sequence of an organism is called its genome.

Genomes represent the fundamental unit of any living organism; nearly every cell of an organism has a complete copy of DNA that makes up the entire organism. In genomes, information is stored about how the entire organism came into being, e.g. how and when a cell differentiates into a certain tissue and how and when a protein is built. The sequence of nucleotides determines the meaning of the information encoded in a particular part of the DNA. Changes in the nucleotide sequence, sometimes even just a single change such as an A instead of a G, can impact the organism significantly. A change in the sequence can cause the protein to malfunction, which in turn can lead to disease, as in the case of hemoglobin, where a single nucleotide substitution can cause hemoglobin to misfold, thus the protein does not function correctly, and causes sickle cell anemia [138].

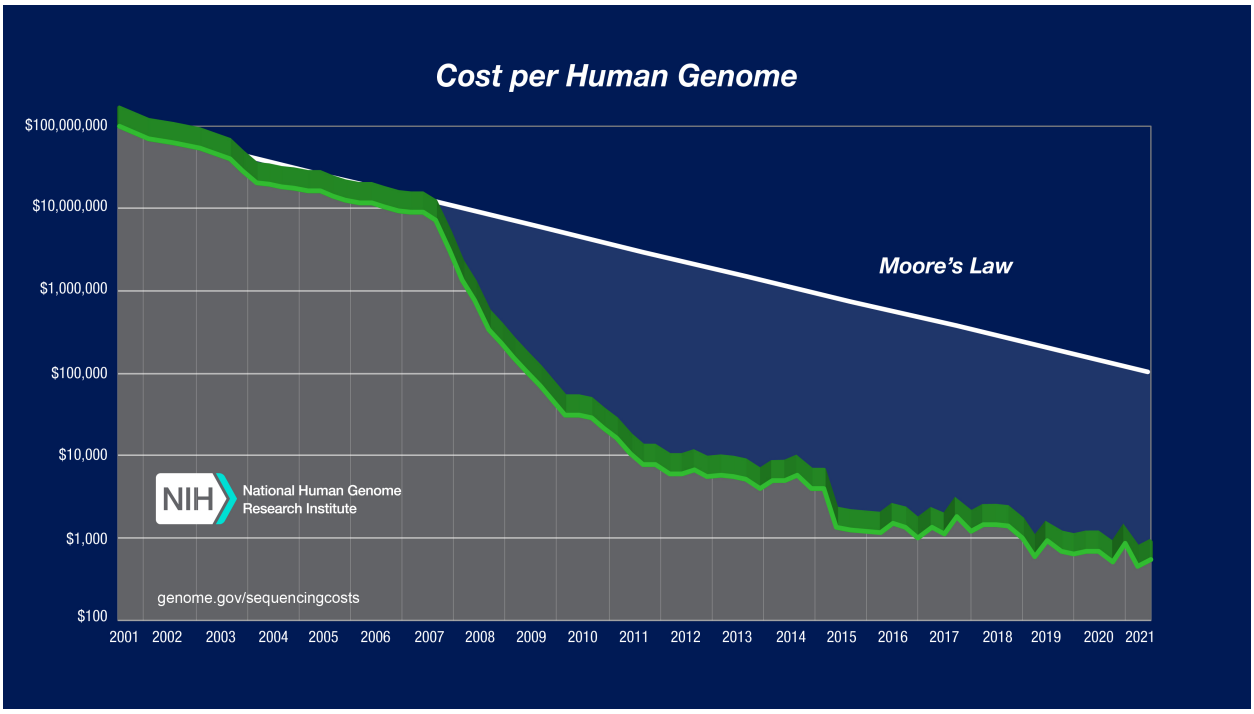

Figure 2.1: Data from the National Institute of Health (NIH) [91] show how the cost of sequencing the human genome has declined over the past decade.

Data from the National Institute of Health (NIH) [91] shown in Figure 2.1 illustrates how the cost of sequencing has dropped significantly over the past decade. This opened the door to a new horizon of real-world scenarios for genomics. Genomics have a primary impact on human health by providing faster and more personalized information about diagnoses and cures for a given patient. In early 2022, Gorzynski et al. [78] demonstrated a record-breaking fast implementation of genome sequencing using optimized sequencing and downstream analysis to diagnose children with suspected genetic disorder who were critically ill and admitted to the intensive care unit—the fastest diagnosis was achieved in 7 hours and 18 minutes.

The impact of genomics goes beyond human health to biofuel production and climate change mitigation [136]. Genomics has the potential to minimize trial and error in agricultural research, improving the quality and quantity of crop production. The ability to link a trait to a gene (a portion of the genome that can store information about the structure of a protein) or a gene signature (a single or combined group of genes in a cell that result in a unique product, such as a protein) would help a plant breeder develop a crop with the best combination of features. Genomics research on plant genomes could show us how to make those genomes more resilient to climate change.

2.2 *De Novo* Long Read Genome Assembly

One of the major computational challenges in high-throughput DNA sequence analysis is *de novo* genome assembly [165]. As we mentioned in Chapter 1, this is the process of aligning and merging redundant, short, and erroneous sequencing reads with the goal of reconstructing the original genome without prior knowledge of, for example, a reference genome for that species. Long-read sequencing technologies [56, 77] yield sequences with an average length of more than 10,000 base pairs (bp). By using longer sequences, we can assemble complex genomic repeats to obtain more precise assemblies that were not possible with short-read technologies [134, 121]. Longer sequences typically come at the price of higher error rates and sequencing cost, leading to increased algorithm complexity and computational cost. Importantly, Pacific Biosciences' latest HiFi technology can deliver long sequences with significantly lower error rates [158].

The Overlap-Layout-Consensus (OLC) paradigm is the most common assembly strategy for long-read sequencing data [16], as we mentioned in the previous chapter. In Chapter 1, Figure 1.1 gives an overview of the OLC paradigm as implemented in this dissertation. The **overlap step** (O) identifies overlapping subsequences between input sequences to create an *overlap* graph (i.e., sequences are vertices and overlapping subsequences between sequences are edges). The idea behind searching for overlapping subsequences is that two sequences that overlap can come from adjacent regions of the genome. Typically, an indexing data structure, such as a k-mer (i.e., a substring of fixed-length k) index table or suffix array, is used to identify an initial set of overlap candidates [103, 100, 16]. Pairwise alignment is sometimes performed to rule out false positives.

The sequencing process introduces redundancy in the input sequences and these also contain inherent genomic repeats that usually make it difficult to interpret and disentangle the overlap graph. For this reason, we perform the **layout step** (L), which simplifies the overlap graph and converts it into a *string* graph (i.e., a graph with no redundant edges and vertices). Importantly, in a string graph, the edges represent the overlap *suffix* rather than the overlap itself, i.e., the region of the sequence that extends beyond the overlap region. The reason is that we want to avoid traversing the overlap region twice when reconstructing the genome sequence using overlapping sequences in the string graph. In general, a string graph can be created from different source graphs depending on the application, not necessarily from an overlap graph. A string graph has the desirable property of collapsing genomic repeats into a single unit [148].

The transitive reduction in the layout step facilitates the clustering of regions of the graph into contigs. In the **consensus step** (C), the string graph is processed to obtain a first draft of the unknown genome, resulting in a set of contigs. A common approach to contig generation uses the string graph as input. Branching nodes in this graph are masked and the set of linear unbranched paths in the graph is extracted to form the contig set. This step can be followed by further polishing phases to merge and correct contigs and to separate haplotypes, i.e. physical groupings of genomic variants that tend to be inherited together.

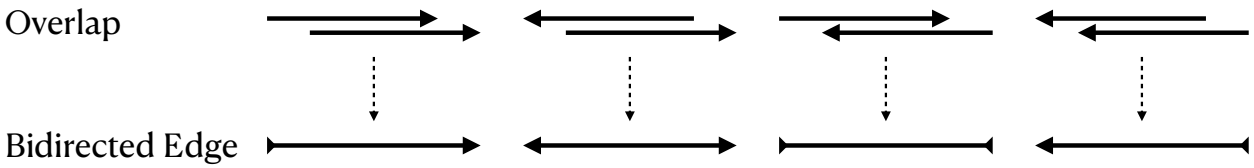

Figure 2.2: Overlap to bidirected edge type mapping.

Overlap Definition For two sequences v_1 and v_2 and their reverse-complements, v_1' and v_2' we can say that v_1 and v_2 have an overlap of length L in base pair (bp) if and only if at least one of these is true:

- The last L bp of v_1 match the first L bp of v_2;
- The last L bp of v_1 match the first L bp of v_2';
- The last L bp of v_1' match the first L bp of v_2;
- The last L bp of v_1' match the first L bp of v_2'.

Considering erroneous sequencing, an overlap in this context indicates that the two sequences have *mostly* identical base content, such that their pairwise alignment score reaches a quality threshold defined by the overlap detection algorithm, i.e., the similarity score between the two sequences must be above 90%.

It is necessary to define four types of overlap, since reads can overlap in a reverse–complement manner and algorithms typically store only the canonical form of k-mers. Additionally, we can define a *contained overlap* as an overlap where the overlapping region of one read is the entire read. An overlap can be called reverse–complement if and only if one of the sequences in the overlap is used in the original direction and the other one is used in the reverse–complement direction. Therefore, an overlap that belongs to the last case is not a reverse–complement overlap, since it is equivalent to the last L bp of v_2 matching the first L bp of v_1. Since the two forward cases are equivalent in theory, we only have three different cases. However, it makes sense to keep the four categories in practice. The correct use of orientation information is crucial during the second and third stages of the OLC algorithm and is essential for the correctness of the final assembly.

Transitivity Definition A string graph (or matrix) is a graph $G = (V, E)$, where V is the set of sequences and E is the set of overlap *suffixes* between any two vertices. There exists an edge if and only if the respective reads overlap and the weight of this edge is the length of the suffix. For example, for the sequences v_1 = TACGA and v_2 = ACGACC, their overlap suffix or *overhang* is the portion of v_2 that exceeds the overlap between v_1 and v_2, i.e. e_{12} = CC. Given $G = (V, E)$, where $V = \{v_1, v_2, v_3\}$ and $E = \{e_{12}, e_{13}, e_{23}\}$, we can walk (a) $v_1 \rightarrow v_2 \rightarrow v_3$ using e_{12} and e_{23}, (b) or $v_1 \rightarrow v_3$ using only e_{13}. If we take the

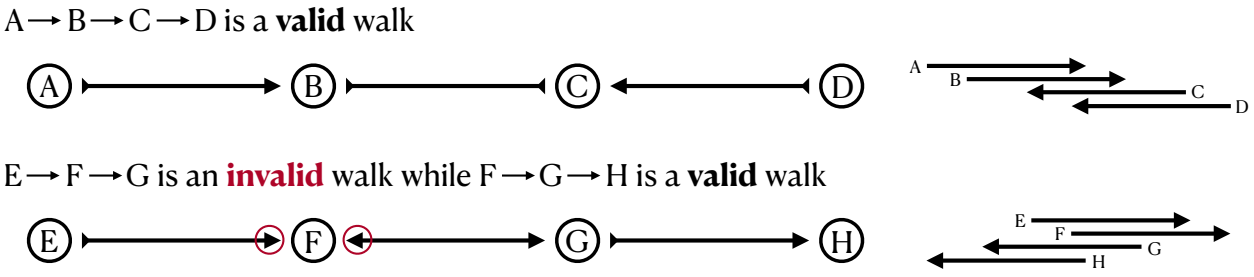

Figure 2.3: An example of a valid and invalid walk in a bidirected string graph.

weight of the edges into account (i.e., the overhang length), we can see that one of these two paths carries less information than the other and therefore we can mark it as *transitive* and remove it from the string graph. This relation is true because contained sequences, i.e., sequences that are fully contained in other sequences, are removed before transitive reduction. Therefore, the two-hop path from $v_1 \rightarrow v_3$ over v_2 stores greater overlap coverage than the direct path $v_1 \rightarrow v_3$. The transitivity of an edge is determined by the coverage of the overlapping region between the sequences under consideration. A larger overlap coverage carries more information, so we want to maintain it over a less informative path.

In *de novo* genome assembly, we want to keep as many overlapping bases as possible for any pair of sequences, so we mark the edges (belonging to a valid path) with longer suffixes (higher weight) as *transitive*. Since the string graph maximizes the overlap length, it can disambiguate short repeats [148]. In our example, e_{13} would be marked as *transitive* and removed since the path $e_{12} \rightarrow e_{23}$ encloses more overlapping bases.

Given we do not know from which strand a certain sequence originates, we want to be able to traverse our graph in both forward and reverse direction. That is, if we consider $G = (V, E)$ in our example above, we want to be able to walk both $v_1 \rightarrow v_2 \rightarrow v_3$ and $v_3' \rightarrow v_2' \rightarrow v_1'$. Using the directed graph representation requires doubling the number of nodes, because for each read we need one vertex representing its *entrance* and one representing its *exit*. Using an undirected graph can avoid this bloat, but it does not guarantee that a particular read will be used in a consistent manner at any point within a single assembly. The use of a bidirected graph (i.e., a graph with a directional head at each end of each edge) [55] solves both of these problems. Figure 2.2 shows the four types of bidirectional edges that result from the four overlap types described earlier.

If $G = (V, E)$ is a bidirected graph, then a valid path in G is a continuous sequence of edges where each vertex is entered by a head inward and exited by a head outward (unless it is the end of the path) or vice versa. Figure 2.3 shows two examples of a valid walk ($A \rightarrow B \rightarrow C \rightarrow D$ and $F \rightarrow G \rightarrow H$) and one example of an invalid walk ($E \rightarrow F \rightarrow G$). A walk through the bidirected string graph encodes the way the sequences can be consistently assembled [119].

Given two paths $v_1 \rightarrow v_2 \rightarrow v_3$ and $v_1 \rightarrow v_3$ in a bidirected string graph, the edge

$v_1 \rightarrow v_3$ can only be considered to be transitive if the following conditions are satisfied: (a) $v_1 \rightarrow v_2 \rightarrow v_3$ constitutes a valid walk, (b) the two heads next to v_1 have the same orientation, and (c) the two heads next to v_3 also have the same orientation.

By definition, a string graph can be constructed from various sources, such as an overlap graph (as we present in this paper), k-mers [95] or FM index [19], and Burrows–Wheeler transform (BWT) [148]. These approaches are not invariant to the input properties and often only consider error-free sequences. In reality, long read data with its high error rates and long lengths often make string graph construction impractical for approaches other than those based on overlap graphs.

2.3 The Combinatorial BLAS Library

Developing high-performance large scale software requires a non-trivial amount of effort and human expertise, such that the effort would be prohibitive for any application that would benefit from a high-performance implementation on a massively parallel machine, such as a supercomputer. A clever solution to this problem is to identify common computational primitives that can cover and represent a wide range of applications, so that we can use a higher level of abstraction of parallel computing [6]. A successful example of primitives that allow the development of much high-performance numerical linear algebra software is the Basic Linear Algebra Subroutines (BLAS) library [102].

Genomics, however, can only be sporadically abstracted using numerical linear algebra, since the most common type of data to be processed is usually a string and its associated metrics, such as how two strings compare. In genomics, and more generally in computational biology, the goal is often to figure out how data relates to each other, e.g. how similar a genome is to genomes in a database or a protein is to other protein sequences, or to find the relationship within a given input dataset, as is the case in *de novo* genome assembly, where the first part of the computation focuses on finding meaningful similarities of sequences to each other and then using that information to reconstruct an unknown genome sequence.

One data representation that is particularly useful for finding the relationship between a set of entities is the graph. Graph algorithms are popular and widely used in computational genomics. However, they are often developed for a specific application, and it is not uncommon to find multiple pieces of software for the same application that implement slightly different heuristics with significantly different codes [103, 100, 70]. This is the case because genomics relies heavily on shared-memory parallelism, which requires less programming effort. However, this status quo is in contrast with the growing demand for scalable parallel software in computational genomics, especially as the amount and type of data continues to increase and performance and scalability become a necessity rather than a luxury.

A major contribution of this dissertation is to consider the *de novo* genome assembly problem, which comes in many shapes in the literature, at a higher level of abstraction that enables flexible and high-performance implementation. In this dissertation, the assembly problem is viewed through the lens of linear algebra, starting from the fact that a graph

can be represented as a sparse matrix. Computation between sparse matrices has long been studied in the parallel computing literature, so we can take advantage of many years of optimized parallel strategies.

Just as numerical linear algebra takes advantage of BLAS, this dissertation on parallel computation for *de novo* long-read genome assembly takes advantage of Combinatorial BLAS (CombBLAS) [29]. CombBLAS is a powerful collection of linear algebra primitives targeted at graph and data mining applications. It enables fast implementation of graph algorithms in distributed memory using a subset of linear algebra operations and the semiring abstraction. The latter provides a way to overload standard linear algebra operators, such as multiplication and addition when performing the dot product between two matrices, with operators useful for the application under consideration. An example is the tropical semiring, where the operations minimum (or maximum) and addition replace the usual operations of addition and multiplication, respectively [135]. The semiring abstraction is critical for genomics because the information stored in a vertex or edge of the graph is usually a complex data structure rather than an integer or floating point number, and we need to be able to model the computation according to the end goal.

The main data structure in CombBLAS is a distributed sparse matrix, but there is also a dense matrix, and a dense vector. Graph computation is implemented as a computation on these data structures. In CombBLAS, the concept of sparse matrix is decoupled from their implementation, so any improvement to the storage format in the literature can be adopted seemingly effortlessly without requiring changes to the entire library.

In terms of computation, CombBLAS has four design principles:

1. If multiple operations can be handled by a single function prototype without affecting the asymptotic performance of the associated algorithm, a generalized single prototype is provided. Otherwise, CombBLAS provides multiple prototypes.

2. If an operation can be implemented efficiently by putting together some simpler operations, then CombBLAS does not provide a special function for that operation.

3. In CombBLAS, many functions also have in-place versions to avoid expensive object creation and copying. For operations that can be implemented in-place, access to other variants is denied only if they increase runtime.

4. In-place operations have slightly different semantics depending on whether the operands are sparse or dense. In particular, the semantics favor preserving the sparsity pattern of the underlying object as long as some other function (possibly not in-place) handles the more conventional semantics that introduces or deletes nonzeros.

In CombBLAS, the implementation of the computation remains largely hidden from the programmer. They must design only the high-level computation as a computation between sparse matrices and use the semiring abstraction as needed. The CombBLAS implementation is written in C++ and is based on the Message Passing Interface (MPI) programming model.

It also takes advantage of hybrid hierarchical parallelism by exploiting flexible shared memory parallelism via the OpenMP API.

CombBLAS assumes that the number of processes is a perfect square. The processes are logically organized as a two-dimensional grid to limit most communication along a processor column or row with $\sqrt{p}$ processes, rather than potentially communicating with p processes. Partitioning distributed matrices (sparse and dense) observes this processor grid organization using a 2D block decomposition, also known as checkerboard partitioning [81].

In the next section, we provide an overview of the Sparse Scalable Universal Matrix Multiplication Algorithm (Sparse SUMMA), which is the algorithm implemented in CombBLAS for Sparse Generalized Matrix Multiplication (SpGEMM) over a general semiring, i.e., the most common and computationally intensive operation in ELBA.

Sparse Generalized Matrix Multiplication (SpGEMM)

In scientific computing and numerical analysis, a *sparse* matrix is defined as a matrix $m \times n$ in which most elements are zero, although a strict ratio is not generally defined. The ratio between the number of zero elements and the total number of elements is called the *sparsity* of the matrix. Conversely, a matrix in which most elements are nonzero is called a *dense* matrix. To process and store sparse matrices, it is preferable and often necessary to develop custom algorithms and data structures, since standard approaches based on dense matrices would result in inefficient or infeasible computation and waste memory due to the large number of zero elements. For these reasons, computations are performed using sparsity by avoiding loading, storing, and computing zero elements.

Compressed Sparse Column and Doubly Compressed Sparse Column Depending on the main purpose of the algorithm, different compressed data structures can be used to store sparse matrices: (a) structures that allow efficient creation and modification of matrices, such as Coordinate List (COO), and (b) structures that support efficient access and matrix computation, such as Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) [143]. In this dissertation, we focus on CSC and on Doubly Compressed Sparse Column (DCSC), a further compressed variant of CSC [25]. The CSC format compresses the column indexes. It uses three 1-dimensional vectors to represent a matrix: (i) a vector NUM that stores the nonzeros, (ii) a vector IR that stores the row indices of the nonzeros, and (iii) a vector JC that stores the indices in NUM and IR where columns begin. The space complexity of CSC is $O(n + nnz)$, where nnz is the number of nonzeros in the matrix and n is the column dimension, since the JC array has size $n + 1$, while IR and NUM have size nnz. CSC provides compression over dense matrix storage structures, but redundant information can still be stored in JC since we can have columns that are completely zero. The repetitiveness can be particularly wasteful for *hypersparse* matrices, i.e., when the number of nonzeros is smaller than the matrix dimension. Such matrices are rare in numerical linear algebra, but they occur frequently in graph computation, which is the focus of our work.

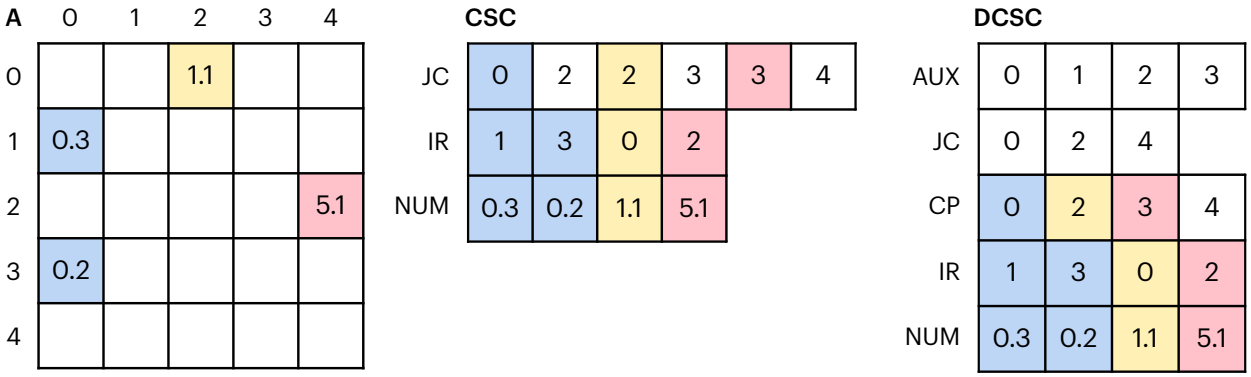

Figure 2.4: The sparse matrix $\mathbf{A}$, its representation in CSC format, and its representation in DCSC format.

Buluç and Gilbert introduced the DCSC format to eliminate the repetitiveness in JC and make it a more efficient alternative for hypersparse matrices [25]. The information-theoretic solution they propose is to compress the JC array by defining nzc as the number of columns that contain at least one nonzero element. If JC is compressed at a rate of $cf = (n+1)/nzc$, it is possible to obtain a compressed array of size nzc. By removing the empty columns in JC, we eliminate the n term in the storage complexity. However, removing the empty columns introduces an indexing problem, since accessing the first element in a column is no longer a constant-time operation. To address this problem, Buluç and Gilbert introduce an additional array called AUX of size $(n+1)/cf \approx nzc$, which stores a pointer to each nonzero column (i.e., column with at least one nonzero). JC is then effectively split into multiple chunks of size $\lceil cf \rceil$, and AUX stores exactly one element per chunk pointing to the first nonzero column in that chunk, plus an additional element at the end that is a null pointer. On average there is one element per chunk, but in the case of a skewed distribution it is possible to have up to $\lceil cf \rceil$ nonzero columns in a single chunk. To efficiently search for a column within a chunk, column indexes are stored in JC, while column pointers (pointing to IR) are stored in a new array called CP as "column pointer". Note that AUX is optional and is not created by default. Figure 2.4 illustrates from left to right: an example matrix $\mathbf{A}$, its representation in CSC format, and its representation in DCSC format.

Classic SpGEMM Given two sparse rectangular matrices $\mathbf{A} \in \mathbb{S}^{m \times k}$ and $\mathbf{B} \in \mathbb{S}^{k \times n}$ from a semiring $\mathbb{S}$, the SpGEMM problem is to compute $\mathbf{C} = \mathbf{AB}$, where $\mathbf{C} \in \mathbb{S}^{m \times n}$ is also a sparse matrix. The input and output matrices are represented in a space-efficient format like the CSC format described above. The number of nonzeros in a matrix $\mathbf{A}$ is written as $nnz(\mathbf{A})$; if the matrix is clear from the context we only write nnz. For the sparse matrix indexing, we use the colon notation used in MATLAB, where $\mathbf{A}(:, j)$ denotes the jth column of $\mathbf{A}$, $\mathbf{A}(i, :)$ denotes the ith row, and $\mathbf{A}(i, j)$ denotes the nonzero at the (i, j)th position of $\mathbf{A}$ (zeros are not stored).

Algorithm 1 Columnwise formulation of serial matrix multiplication [88].

```
1: procedure COLUMNWISE-SPGEMM(A, B, C)
2:     for j ← 1 to n do
3:         for k where B(k, j) ≠ 0 do
4:             C(:, j) ← C(:, j) + A(:, k) · B(k, j)
5:         end for
6:     end for
7: end procedure
```

The classical serial multiplication algorithm for general sparse matrices was first described by Gustavson [88] and subsequently implemented in MATLAB [74] and CSparse [48]. CSC was the data structure used by Gustavson to store sparse matrices and is also used in most sparse matrix packages, including MATLAB. Algorithm 1 gives the pseudocode for the column-wise serial algorithm described by Gustavson [88].

The number of nonzero arithmetic operations between $\mathbf{A}$ and $\mathbf{B}$ required to compute the final matrix $\mathbf{C}$ is called "flops". The computational complexity of a sparse matrix algorithm should preferably depend on this number. The computational complexity of sparse multiplication in Gustavson's and MATLAB's original design is $O(flops + nnz(\mathbf{B}) + n + m)$, i.e., it is proportional to the number of nonzero elements and linearly dependent on the row size m and column size n of the matrix. The sparse implementation of MATLAB defines an abstract data structure, the sparse accumulator or SPA, that provides random access to the currently "active" column or row of a matrix. Typically, sparse matrix algorithms use a dense working vector to provide fast, random access to the currently "active" column or row of a matrix. SPA has three components: (i) a dense vector of real values for the active column of $\mathbf{C}$, (ii) a dense Boolean vector of "occupied" flags, and (iii) an unordered list of indices whose occupied flag is true. An index whose occupied flag is true represents the indexes of the nonzeros of the currently active column or row. The SPA has a space complexity of $O(m)$ and its initialization has a time complexity of $O(m)$, contributing to the m-term in the complexity of the MATLAB algorithm. Moreover, the n-term in the complexity formula is derived from the column-wise implementation. Figure 2.5 illustrates this algorithm.

Distributed SpGEMM An important aspect of implementing sparse matrix multiplication in distributed memory is how to store the matrices across processes. Each process stores only a subset of the matrices and there are two common types of matrix distribution: 1-dimensional and 2-dimensional. In 1D algorithms, each process stores a block of m/P rows of an m-by-n sparse matrix, where P is the total number of processes. In 2D algorithms, the processes are logically organized as a rectangular $P = P_r \times P_c$ grid, so a typical process is called $P(i, j)$. The submatrices are assigned to the processes according to a 2D block decomposition: process $P(i, j)$ stores the submatrix $\mathbf{A}_{ij}$ of dimensions $(m/P_r) \times (n/P_c)$ in its local memory. We extend the colon notation to slices of submatrices: $\mathbf{A}_{i:}$ denotes the

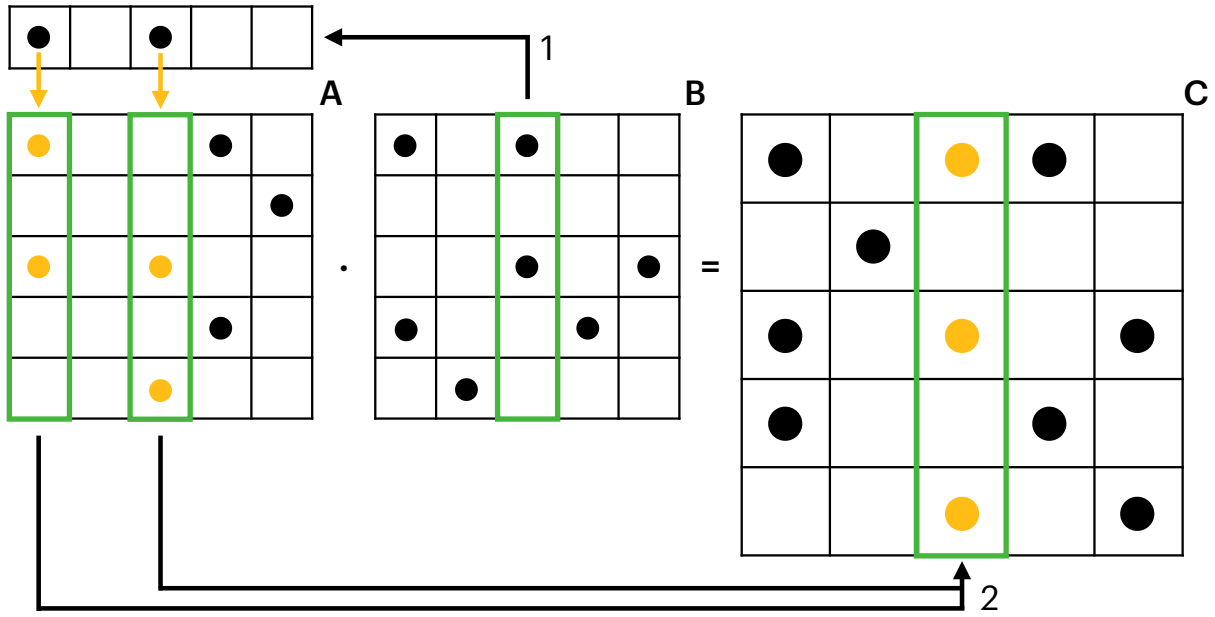

Figure 2.5: Column-by-column SpGEMM [88, 74, 48].

$(m/P_r) \times n$ slice of $\mathbf{A}$ common to all processes along the ith process row, and $\mathbf{A}_{:j}$ denotes the $m \times (n/P_c)$ slice of $\mathbf{A}$ common to all processes along the jth process column. The literature has shown that 1D algorithms are not scalable for thousands of processes, mainly because of the non-scalable communication cost [24]. Therefore, we consider only 2D algorithms here.

Previously, we introduced the CSC format and mentioned that CSC is the standard format used in MATLAB and other libraries when performing SpGEMM. However, Buluç and Gilbert [25] have shown that CSC is too wasteful for storing matrices over a 2D process grid because the local submatrices are hypersparse, meaning that the number of nonzeros is smaller than the matrix dimension. If we were to use CSC in distributed memory, the total memory for all processes would be $O(n\sqrt{P} + nnz)$, while for a single process it would be $O(n + nnz)$. Thus, a scalable parallel 2D data structure must take hyperspareness into account, and the DCSC format presented earlier achieves this goal by having a memory requirement of $O(nnz)$ regardless of matrix dimension.

As for the storage format, when implementing the sparse multiplication algorithm, we also want to consider hyperspareness and eliminate the dependence on the matrix dimension. Gustavson's serial SpGEMM algorithm would be too wasteful for hypersparse submatrices. To eliminate the dependence on matrix dimension, Buluç and Gilbert introduced the Hyper-SparseGEMM algorithm [28, 25], which uses an outer product formulation and the DCSC format. HyperSparseGEMM has a time complexity of $O(nzc(\mathbf{A}) + nzr(\mathbf{B}) + \text{flops} \cdot \lg(ni))$, where $nzc(\mathbf{A})$ is the number of nonempty columns of $\mathbf{A}$, $nzr(\mathbf{B})$ is the number of nonempty rows of $\mathbf{B}$, flops is the number of nonzero arithmetic operations between $\mathbf{A}$ and $\mathbf{B}$ required to compute the final matrix $\mathbf{C}$, and ni is the number of indices i for which $\mathbf{A}(:, i) = \varnothing$ and

Algorithm 2 Distributed SpGEMM using Sparse SUMMA [26, 24].

1: **procedure** SPARSESUMMA($\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$)
2: **for** all processes $P(i, j)$ **in parallel do**
3: $B_{ij} \leftarrow (B_{ij})^{\mathsf{T}}$
4: **for** q = 1 to k/b **do** ▷ blocking parameter b evenly divides k/P_r and k/P_c
5: $c = (q \cdot b)/P_c$ ▷ c is the broadcasting processor column
6: $c = (r \cdot b)/P_r$ ▷ r is the broadcasting processor row
7: $lcols = (q \cdot b) \bmod P_c : ((q + 1) \cdot b) \bmod P_c$ ▷ local column range
8: $lrows = (q \cdot b) \bmod P_r : ((q + 1) \cdot b) \bmod P_r$ ▷ local row range
9: $\mathbf{A}^{rem} \leftarrow$ BROADCAST($\mathbf{A}_{ic}(:, lcols), P(i, :)$)
10: $\mathbf{B}^{rem} \leftarrow$ BROADCAST($\mathbf{B}_{rj}(:, lrows), P(:, j)$)
11: $\mathbf{C}_{ij} \leftarrow \mathbf{C}_{ij}$ + HYPERSPARSEGEMM($\mathbf{A}^{rem}, \mathbf{B}^{rem}$)
12: **end for**
13: $B_{ij} \leftarrow (B_{ij})^{\mathsf{T}}$ ▷ restore the original B_{ij}
14: **end for**
15: **end procedure**

$\mathbf{B}(i, :) = \varnothing$. The factor $\lg(ni)$ comes from the priority queue used to merge ni outer products on the fly. The total memory requirement of this algorithm is $O(nnz(\mathbf{A})+nnz(\mathbf{B})+nnz(\mathbf{C}))$, i.e., independent of matrix dimension.

Buluç and Gilbert [26, 24] have proposed a parallel algorithm for multiplying sparse matrices that is similar to the algorithm implemented in BLAS [102] for multiplying dense matrices, i.e., SUMMA [155], in that it is memory efficient and can be easily generalized to non-square matrices and processes grid. This parallel algorithm is called SparseSUMMA, uses HyperSparseGEMM as kernel, and operates over a 2D processor grid. The pseudocode is shown in Algorithm 2, which for simplicity illustrates the case where we have a square process grid. For a rectangular process grid, a given processor row and column would potentially require more than one broadcast operation during an iteration of the loop starting at line 4. 3D SpGEMM algorithms that are faster than the 2D SparseSUMMA algorithm exist both in the literature and in CombBLAS [10], but most of the other functions in CombBLAS that we use in this dissertation operate on a 2D grid, so it is not advantageous for us because of the conversion cost between 3D and 2D. A 2D algorithm like SparseSUMMA is based on a 2D decomposition of the output matrix and the computation follows the "owner computes" rule, while a 3D algorithm also parallelizes the computation of the individual entries of the output matrix [10].

Parallel processes broadcast their local submatrix of $\mathbf{A}$ to their entire process row and likewise they broadcast their local submatrix of $\mathbf{B}$ to their entire process column. In this way, each process has all the entries necessary to compute its local submatrix of the final matrix $\mathbf{C}$ using the HyperSparseGEMM algorithm. $\mathbf{B}$ is transposed before broadcasting so that it can be indexed column-wise, since the local submatrices are stored in column-based

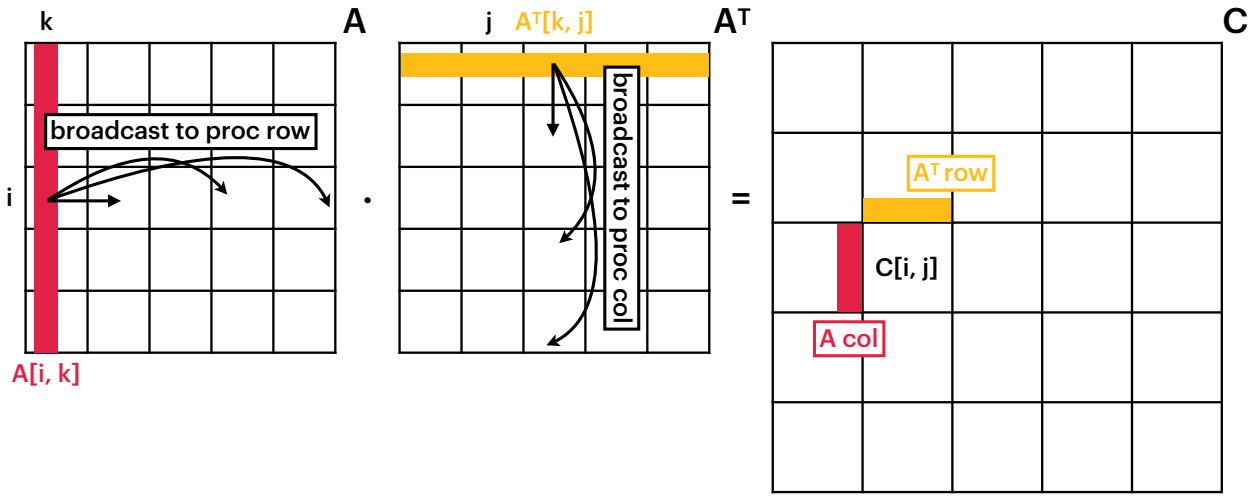

Figure 2.6: The Sparse SUMMA algorithm for sparse matrix-matrix multiplication $\mathbf{A} \cdot \mathbf{A}^{\mathsf{T}} = \mathbf{C}$ [26]. The example shows the first stage of the algorithm execution (the broadcast and the local update by processor $P(i, j)$ of the block $\mathbf{C}(i, j)$ in the output matrix $\mathbf{C}$).

DCSC sparse format and column-wise indexing of $\mathbf{B}$ is faster. The execution of the algorithm is illustrated in Figure 2.6.

Chapter 3

Parallel Algorithms for Overlap Detection

In this chapter, we describe the mathematical methodologies we have developed and the parallel algorithms we have designed for the overlap detection phase of *de novo* long-read genome assembly. In particular, we first describe the Markov chain model that we developed to prove the feasibility of a seed-based approach for *de novo* long-read assembly and to find the optimal k-mer length based on the dataset. Then, we describe a method for pruning the k-mer space to remove unwanted or uninformative information, and a method for distinguishing between true and false overlap matches based on the alignment score. Finally, we describe our parallel algorithms for overlap detection using sparse matrices and semiring abstraction, and report experimental results in terms of quality and runtime performance.

3.1 Overview and Foundation

Overlap detection is usually the first step in Overlap-Layout-Consensus (OLC) assembly, the predominant method for *de novo* assembling long read data [16, 109]. A read-to-read overlap is a sequence match between two sequences to determine whether local areas on each read originate from the same location within a larger sequence. OLC uses these matches to create an overlap graph where each node is a read and each edge is an overlap connecting them. Overlap detection has been shown to be a major bottleneck in efficiency when using the OLC assembly method [120] for large genomes. Overlap detection is used not only as the first step in the OLC assembly paradigm, but also as the first step in pipelines whose goal is to error-correct sequences before other computation [41].

In the literature, we find several software packages that perform overlap detection with different accuracy and runtime performance. Despite using different strategies, they share some common features, such as using short subsequences (i.e., seeds) to discover overlap candidates. The main differences between the algorithms are not only in the way common seeds are found, but also in the way the seeds are used to determine whether an overlap candidate is correct or not. It is common to prune the seed space using different methodolo-

gies [100, 107, 34]. Once a method finds the candidates, it validates them and computes the estimated overlap areas by comparing the coordinates of each common seed. Each method generates a list of overlap candidates and provides an overlap region between sequences. In some pipelines, most of the computational time is spent aligning candidates at the nucleotide level, either for error correction [150] or for weeding out spurious candidates after they have been found. In other pipelines, nucleotide-level alignment may not be used but replaced by an approximation method using only seed coordinates [103]. Therefore, the output of any overlap algorithm includes at least the overlap coordinates and often some additional information for downstream analyzes, such as alignment scores or sequence similarity estimates.

ELBA follows the literature and uses a seed-based approach for overlap detection. In particular, ELBA uses an exact seed approach where a seed is called k-mer, i.e., a substring of fixed length k. Using a Markov chain model, we first demonstrate the feasibility of a k-mer seed-based approach to long read overlap detection. The descriptiveness of our model enables us to model the probability of finding a correct shared k-mer between two sequences, even when using different seed strategies than ours, such as minimizer [103] and syncmer [54]. This model also provides us with the optimal k-mer length, which is affected by the error rate and sequencing coverage of the data, as well as the desired level of detection accuracy.

Then, we describe how ELBA chooses the k-mer set to identify overlap candidates using a novel probabilistic method to filter out those that are likely to be either erroneous or from a repetitive region. The k-mer set retained by ELBA is considered *reliable*, where the reliability of a k-mer is defined as its probability of originating from a unique (non-repetitive) region of the genome. In this context, we argue that unique k-mers are sufficient for overlap detection because long read sequences either (a) contain long enough non-repetitive sections to identify overlaps with unique k-mers, or (b) are entirely contained in a repeat, in which case their overlaps are inherently ambiguous and uninformative. Our reliable k-mer detection maximizes the retention of k-mers from unique regions of the genome using probabilistic analysis at a given error rate and sequencing depth.

Once we have identified the overlap candidates, ELBA computes the nucleotide-level alignment for each of them using the SeqAn seed-and-extend algorithm [53], which tries to extend the match region between two sequences. This algorithm computes the alignment starting from the common k-mer coordinates and by extending the match to the left and right and not necessarily from the beginning of the sequences. During the alignment phase, ELBA uses a new method to distinguish true overlap candidates from false positives as a function of the alignment score. Our method proves that the probability of false-positive candidates decreases exponentially as the length of overlap between sequences increases.

Overlap Feasibility

Chaisson and Tesler [35] proposed a theory of how long read sequences contain subsequences that can be used to anchor alignments to the reference genome. The sequences are modeled as random processes that generate error-free regions whose length is geometrically distributed,

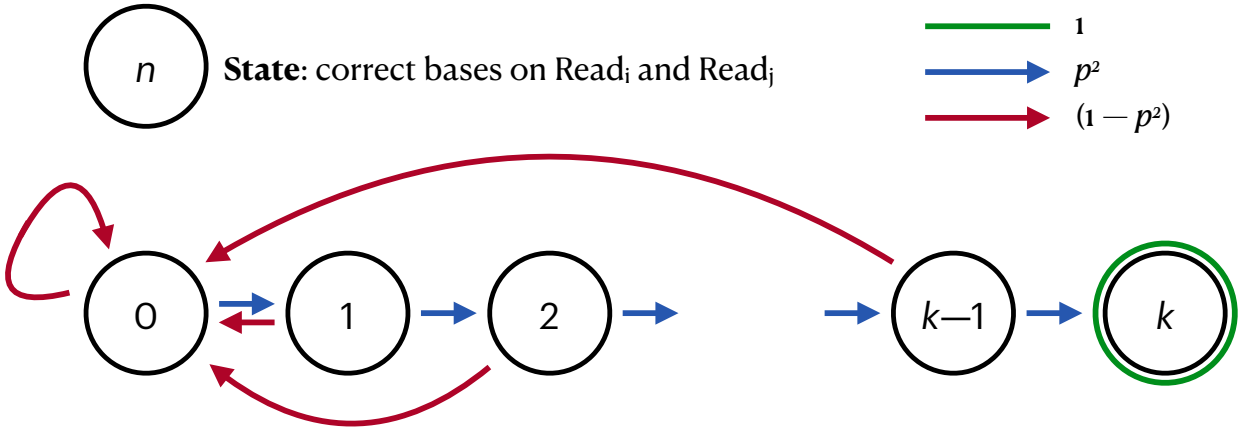

Figure 3.1: Proposed Markov Chain model demonstrating the feasibility of using k-mers for overlap detection.

with each such region separated by an error [75]. The result of their theory is the minimum sequence length to have an *anchor* within a confidence interval.

Here we present an alternative model of how these subsequences, also called k-mers, can be used to anchor alignments between two erroneous long read sequences, allowing accurate overlap detection between all reads in a dataset. The original assumption of our model defines the probability of correctly sequencing a base as equal to $p = (1 - e)$, where e is the error rate of the sequencer. Based on this notion, we model the probability of observing k correct consecutive bases on both $read_1$ and $read_2$ as a Markov chain process [110].

The Markov chain process is characterized by a *transition matrix* $\mathbf{P}$ containing the probabilities of transition from one state to another. Each row index *start* of $\mathbf{P}$ represents the initial state and each column index *end* of $\mathbf{P}$ represents the final state. Each entry of $\mathbf{P}$ is a non-negative number indicating a *transition probability*. Our transition matrix has $(k + 1)$ possible states, resulting in $(k + 1)^2$ transition probabilities for the transition from *start* to *end*. The probability of having a correct base in both reads is p^2. For every state except the *absorbing* state k, an error in at least one of the two sequences causes the model to fall back to state 0, which happens with probability $1 - p^2$; otherwise, the transition of the Markov chain from state i to $i + 1$ happens with probability p^2. The absorbing state k cannot be abandoned because both $read_1$ and $read_2$ have already seen k successive correct bases. Therefore, its transition probability is 1. Figure 3.1 describes the process: each state contains the number of successfully sequenced bases obtained on both reads up to that point, while the arrows represent the transition probabilities.

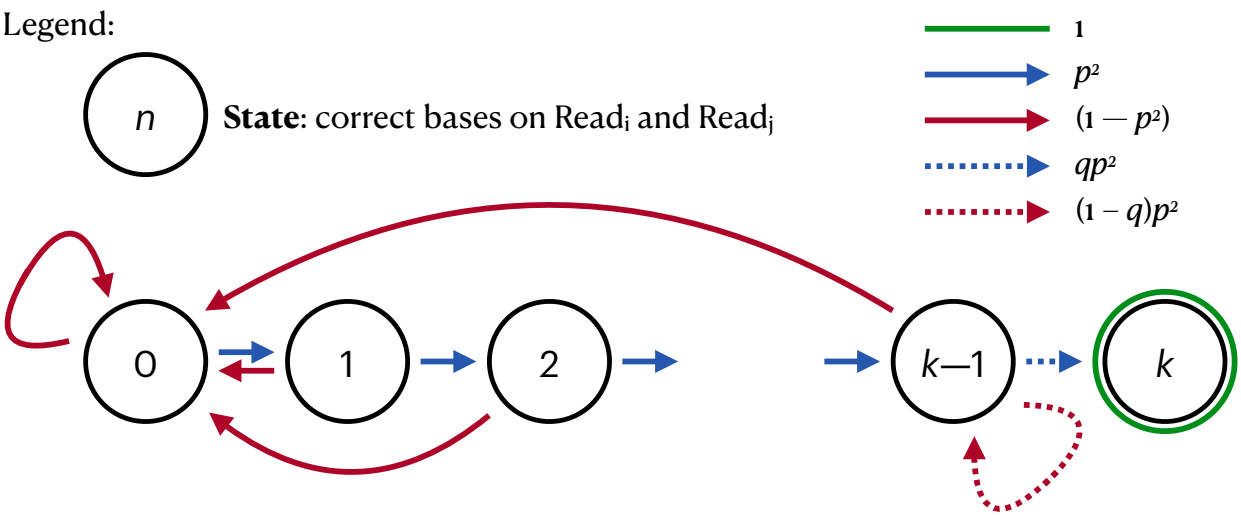

Figure 3.2: Proposed Markov Chain model using syncmer approach instead of the k-mer one, where $q = 1/c$.

The probability of being in one of the states after L steps in the Markov chain can then be determined by computing the L-th power of the matrix $\mathbf{P}$, where L is the length of overlap between the two sequences. More efficiently, this can be computed iteratively with only L sparse matrix vector products, starting from the unit vector $\mathbf{v} \leftarrow (1, 0, \ldots, 0)$ (Algorithm 3). This approach is sufficient since we are only interested in the probability of being in the absorbing final state. This operation leads to the probability of a correct k-mer being in the same place in both reads, given a certain overlap region. This model is the driving factor for choosing the optimal k-mer length used in overlap detection.

Our Markov chain can be modified to account for different k-mer selection strategies, such as syncmers and minimizers. Given a compression factor $c > 1$, which sets a minimum value for the k-mer code, a k-mer κ is a *mincode syncmer* if $code(\kappa) \leq H/c$, where H is the maximum possible code [54]. The probability that a given k-mer is selected as a mincode syncmer is $1/c$. In this case, we can modify our model to include the probability that a k-mer is correct *and* that it is retained as a mincode syncmer. The transition from the $(k-1)$-th state to the k-th state in the Markov chain will model syncmer selection such that we have a probability of transitioning from $k-1$ to k that is equal to qp^2, where $q = 1/c$, i.e., the k-th is correctly sequenced and the k-mer is a mincode syncmer. Then we have a probability of $(1-q)p^2$ to stay in the $(k-1)$-th state, which means that we have sequenced a correct base for both sequences, but the k-mer is not selected as a syncmer. The probability of returning to the initial state is unchanged. The Markov chain model modified for mincode syncmer is shown in Figure 3.2.

In the figure, we have used the mincode syncmer as an example, but the same proba-

Algorithm 3 Probability of observing at least one correct k-mer in an overlap region of length L > k.

```
1: procedure EstimateSharedKmerProbability(k,L,p)
2:     states ← (k + 1)
3:     P ← 0                                   ▷ Entire matrix initialized to 0
4:     for i ← 0 to states do
5:         P[i, 0] ← (1 − p^2)
6:         P[i, i + 1] ← p^2
7:     end for
8:     P[states, states] ← 1
9:     v ← (1, 0, ..., 0)                      ▷ Initialized to standard unit vector
10:    for i ← 0 to L do                       ▷ Compute vP^L without exponentiation
11:        v ← vP
12:    end for
13:    return v[states]
14: end procedure
```

bilistic model applies to other syncmer types, including those with better spacing properties, such as closed syncmer [54]. This is because the selection of a k-mer as a syncmer is by definition a local decision and is not influenced by neighboring k-mers. The case of the minimizer is slightly different. A k-mer is a minimizer if it has the smallest code among w successive k-mers, where w is called the window length [141]. For a k-mer κ that is correctly sequenced in both reads, we must consider the number of competing k-mers in its windows to determine the probability that κ is selected as the minimizer *from both reads*. If there are no sequencing errors, there are w competing k-mers including κ itself. Since errors can change the competing k-mers in each read independently, the maximum number of competing k-mers including κ itself is $2w - 1$. It is possible to calculate the exact expected number of competing k-mers, but since this range is narrow within a factor of two, in practice we can also use the upper and lower bounds when choosing the minimizer parameter (w, k).

Reliable k-mer Set

Repetitive regions of the genome cause certain k-mers to occur frequently in input sequences. K-mers from these regions pose two problems for pairwise overlap and alignment. First, their presence increases computational costs, both in the overlap and alignment phases, because these k-mers generate numerous and possibly incorrect overlaps. Second, they often do not provide valuable information.

Here we argue that k-mers originating from repetitive regions can be ignored for seed-based overlap. This is because either (a) the read is longer than the repeat, in which case there should be enough sequence data from the non-repetitive section to find overlap candidates, or (b) the read is shorter than the repeat, in which case their overlap matches are

inherently ambiguous and uninformative and not particularly useful for *de novo* assembly. In the case of a nearly identical region, we would expect to find a k-mer that comes from a unique region of that nearly identical repeat to identify that region.

Following the terminology proposed by Lin et al. [107], we refer to k-mers that are not present in the genome as *non-genomic* and thus characterize k-mers that are present in the genome as *genomic*. A genomic k-mer can be *repeated* if it occurs multiple times in the genome, or *unique* if it does not. One can think of the presence of k-mers in each read as the feature vector of that read. Therefore, the feature vector should contain all unique k-mers, as they are often the most informative features.

Since we do not know the genome before assembly, we estimate the genomic uniqueness of k-mers from redundant, error-prone reads. Here we present a mathematically based procedure that selects a frequency range for k-mers that we consider *reliable*. The fundamental question guiding the procedure for selecting reliable k-mers is the following: "What is the probability that a k-mer occurs at least m times in the input data if it was sequenced from a unique (non-repeat) region of the genome?". For a genome G sequenced at a sequencing depth d or sequencing coverage (i.e., the number of unique sequences containing a given nucleotide in the reconstructed genome sequence), the conditional modeled probability is:

$$Pr(\text{FREQ}(k\text{-}mer, G, d) \geq m \mid \text{COUNT}(\text{MAP}(k\text{-}mer, G) = 1) \tag{3.1}$$

where $\text{MAP}(k\text{-}mer, G)$ is the set of sites in the genome G to which k-mer can be mapped, the function $\text{COUNT}()$ computes the cardinality of a given input set, and $\text{FREQ}(k\text{-}mer, G, d)$ is the expected number of occurrences of k-mer within the sequenced reads, assuming that each region of G is sequenced d times. In this sense, ELBA's approach to selecting reliable k-mers is very different from the way Lin et al. [107] select their *solid strings*. The solid strings exclude rare k-mers, while our model excludes highly recurrent k-mers because (a) unique k-mers are sufficient to find informative overlaps, and (b) a unique k-mer has a low probability of occurring frequently.

The probability of correctly sequencing a k-mer is approximately $(1-e)^k$, where e is the error rate. The probability of correctly sequencing a k-mer once can be generalized to the probability of seeing it multiple times in the data, provided that each correct sequencing of that k-mer is an independent event. For example, if the sequencing depth is d, the probability of observing a unique k-mer K_u in the input data d times is approximately $(1-e)^{dk}$. More generally, the number of correct sequencings of a unique k-long genome segment at a sequencing depth d follows a binomial distribution:

$$B(n = d, p = (1-e)^k) \tag{3.2}$$

where n is the number of trials and p is the probability of success. From this, we derive that the probability of observing a k-mer K_u (corresponding to a unique non-repetitive region of the genome) m times within a sequencing input data with depth d is:

Algorithm 4 Reliable k-mer range: Selection of the lower bound (l)

```
1: procedure l ← LowerBound(d, e, k)      ▷ d: depth, e: error rate, k: k-mer length
2:     sum ← 0                                                    ▷ Cumulative sum
3:     m ← 2                                  ▷ The k-mer multiplicity in the input
4:     while (sum < ϵ) do
5:         probability ← P(d, e, k, m)
6:         sum ← sum + probability
7:         m ← m + 1
8:     end while
9:     l ← m − 1
10:    return l                                                  ▷ The lower bound
11: end procedure
```

$$Pr(m; d, (1-e)^k) = \binom{d}{m}(1-e)^{km}(1-(1-e)^k)^{(d-m)} \tag{3.3}$$

where m is the number of occurrences of a k-mer K_u in the input data, where the input is the genome sequenced at depth d, e is the error rate, d is the sequencing depth, and k is the k-mer length. Given the values of d, e, and k, the curve $Pr(m; d, (1-e)^k)$ can be calculated.

Equation 3.3 is used to determine the range of reliable k-mers. To choose the lower bound l, we calculate $Pr(m; d, (1-e)^k)$ for each multiplicity m and sum these probabilities cumulatively, starting with $m = 2$. The cumulative sum does not start at $m = 1$ because a k-mer that occurs only once in the input data (and therefore appears on a single read) cannot be used to identify the overlap between two reads. The lower bound l is the smallest m value after which the cumulative sum exceeds a user-defined threshold ϵ. The choice of l is significant when the sequencing error rate is relatively low ($\approx 5\%$) or when the sequencing depth or coverage d is high ($\approx 50 - 60\times$), or both. This is because in these cases, a k-mer with small multiplicity has a high probability of being incorrect.

The upper bound u is chosen in a similar way. Here, starting from the largest possible value of m (i.e. d), the probabilities are added up cumulatively. In this case, u is the largest value of m after which the cumulative sum exceeds the user-defined threshold ϵ. The k-mers that are more frequent than u have too low a probability of belonging to a unique region of the genome, and multiple mapped k-mers would lead to an increase in computational cost and possibly misassembly.

K-mers with multiplicity greater than u and multiplicity less than l in the input set are discarded and not used as read features in the downstream algorithm. Our reliable k-mers selection discards at most 2ϵ useful information when the data fits the model, in the form of k-mers that can be used for overlap detection.

If you use the syncmer strategy instead of the k-mer strategy, the reliable range calculations remain unchanged. This is because a given k-mer is either selected as a syncmer

Algorithm 5 Reliable k-mer range: Selection of the upper bound (u)

```
1: procedure u ← UpperBound(d, e, k)      ▷ d: depth, e: error rate, k: k-mer length
2:     sum ← 0                                                    ▷ Cumulative sum
3:     m ← d                                  ▷ The k-mer multiplicity in the input
4:     while (sum < ϵ) do
5:         probability ← P(d, e, k, m)
6:         sum ← sum + probability
7:         m ← m − 1
8:     end while
9:     u ← m + 1
10:    return u                                                   ▷ The upper bound
11: end procedure
```

for all its occurrences in the dataset or is never selected as a syncmer. For minimizers, the exact computation of the reliable range is non-trivial, since errors in flanking sequences affect whether a k-mer is retained as a minimizer. Therefore, we leave this as future work.

Adaptive Alignment Threshold

In outputting overlap candidates, high precision is desirable to avoid unnecessary work at later stages of the assembly. Therefore, ELBA prunes overlap candidates by performing a pairwise seed-and-extend alignment on each candidate pair. Unlike approaches based on sketches, such as minimap2 [103] and MHAP [16], seed-and-extend alignment can be performed directly with the k-mers from the overlap phase of ELBA. The alignment module of ELBA uses the x-drop seed-and-extend alignment of SeqAn [53].

The x-drop defines the termination condition of the alignment, i.e., for each read pair identified as an overlap candidate, the alignment is extended from the k-mer match until the alignment score either reaches the end of one of the two sequences or decreases by x from the previous best score. If the final score is less than a threshold n, the sequence pair is discarded. For this reason, this approach is also called "x-drop alignment". To filter out spurious candidates, we use an *adaptive threshold* n calculated based on the estimated overlap between a given pair of sequences. The choice of the scoring matrix used in the pairwise alignment step justifies that the alignment score threshold is a linear function of the estimated overlap length.

Given an estimated overlap region of length L and probability p^2 of obtaining a correct base in both sequences, we would expect $m = p^2 \cdot L$ correct matches within this overlap region. The alignment score χ is:

$$\chi = \alpha m - \beta(L - m) = \alpha p^2 L - \beta(L - p^2 L) \tag{3.4}$$

where m is the number of matches, L is the overlap length, α is the value associated with

a match in the scoring matrix, while β is the penalty for a mismatch or a gap ($\alpha, \beta > 0$). Under these assumptions, we define the ratio φ as χ over the estimated overlap length L as:

$$\varphi = \frac{\chi}{L} = \alpha p^2 - \beta(1 - p^2). \tag{3.5}$$

The expected value of φ is equal to $2 \cdot p^2 - 1$ when an exact alignment algorithm is used. Thus, we want to define a threshold with respect to $(1 - \delta)\varphi$ such that we keep pairs above this threshold as true alignments and discard the remaining pairs. To this end, we use a Chernoff bound [39, 92] to define the value of δ, and prove that there is only a small probability of missing a true overlap of length $L \geq 2000$ bp (which is typically the minimum overlap length for a sequence to be considered true positive) when we use the threshold defined above.

Chernoff Bound Derivation Let Z be a sum of independent random variables $\{Z_i\}$, with $E[Z] = \mu_z$; we assume for simplicity that $Z_i \in \{0, 1\}$, for all $i \leq L$. The Chernoff bound defines an upper bound on the probability that Z deviates from its expected value by a certain amount δ. In particular, we use a corollary of the multiplicative Chernoff bound [5], which is defined as follows for $0 \leq \delta \leq 1$:

$$\mathbf{Pr}[Z \leq (1 - \delta)\mu_z] \leq e^{\frac{-\delta^2 \mu_z}{2}} \tag{3.6}$$

To obtain the Chernoff bound for the ratio φ, we consider a random variable $X_i \in \{-\beta, \alpha\}$ such that:

$$X_i = \begin{cases} \alpha, & \text{with probability } p^2 \\ -\beta, & \text{with probability } 1 - p^2 \end{cases} \tag{3.7}$$

where $\alpha, \beta > 0$ are still the values associated with a match and a mismatch and a gap in the scoring matrix, respectively; its expected value $E[X_i]$ is exactly equal to $\varphi = \alpha p^2 - \beta(1 - p^2)$. Since the Chernoff bound is defined for a sum of independent random variables $Z_i \in \{0, 1\}$, we need to move from $X_i \in \{-\beta, \alpha\}$ to $Z_i \in \{0, 1\}$. Thus, we define a new random variable $Y_i = X_i + \beta$ as a linear transformation of X_i that can take values $\{0, \alpha + \beta\}$. Using $E[Y_i] = E[X_i] + \beta = (\alpha + \beta)p$, we can normalize Y_i to obtain the desired random variable Z_i:

$$Z_i = \frac{X_i + \beta}{\alpha + \beta}, \text{ where } Z_i \in \{0, 1\} \tag{3.8}$$

From the linearity of expectation, we obtain:

$$E[Z] = E\left[\frac{X + \beta}{\alpha + \beta}\right] = \frac{E[X] + \beta L}{\alpha + \beta} = \frac{(2p^2 - 1)L + \beta L}{\alpha + \beta} \tag{3.9}$$

Finally, substituting Eq. 3.8 and Eq. 3.9 into Eq. 3.6 and simplifying with our scoring matrix $\alpha, \beta = 1$, we get the final expression:

$$\mathbf{Pr}[X \leq (1-\delta)\mu_x] \leq e^{-\delta^2 p^2 L}, \text{ with } E[X] = \mu_x \tag{3.10}$$

For two sequences that correctly overlap by $L = 2000$, the probability that their alignment score is more than 10% ($\delta = 0.1$) below the mean is $\leq 5.30 \times 10^{-7}$. ELBA, with an x-drop value of $x = 50$ and an adaptive threshold derived from the scoring matrix as well as a cutoff rate of $\delta = 0.1$, achieves high values for recall and precision compared to state-of-the-art software. Recall is defined as the number of correct overlap candidates identified by ELBA relative to the total number of correct overlap matches in the input dataset, while precision is the ratio of correct overlap candidates identified by ELBA to the total number of overlap candidates identified by ELBA. This bound can be relaxed, e.g., $\delta = 0.7$, if the error rate in the sequencing data decreases without compromising accuracy performance with the state-of-the-art.

3.2 Proposed Algorithm

Having described the mathematical methodologies used in our algorithms in the previous section, in this section we describe how ELBA actually computes overlap detection.

ELBA introduces sparse matrices to represent its data internally, where the rows are sequences, the columns are the reliable k-mers, and a nonzero $\mathbf{A}(i,j) \neq 0$ is the position of the j-th k-mer within the i-th read. The construction of this sparse matrix requires efficient k-mer counting. Overlap detection is implemented in ELBA using SpGEMM (see Chapter 2), which provides fast overlap detection without using approximate approaches. SpGEMM is an extremely flexible and efficient paradigm that allows for better organization of computations and greater generality because it can manipulate complex data structures, such as those used to perform overlap detection, by using common k-mers. Our k-mer selection can be easily replaced by other selection strategies without affecting SpGEMM-based overlap detection, demonstrating the generality of our approach. The implementation of this method in our pipeline enables the use of high-performance techniques not previously used in the context of long-read alignment. It also enables continuous performance improvements in this step through increasingly optimized implementations of SpGEMM [122, 49]. This abstraction makes software design more flexible and modular, and enables high parallelization that can scale in distributed memory. To implement parallel computation over sparse matrices, we use the Combinatorial BLAS (CombBLAS) library [9], a well-known framework for implementing graph algorithms in the language of linear algebra, and use a semiring abstraction to overload the classical multiplication and addition operations as needed. A description of CombBLAS can be found in Chapter 2.

ELBA's overlap detection has been coupled with SeqAn's seed- and-extend algorithm [53], which computes the alignment starting from the seed coordinates to the left and right of the seed match and tries to extend the match. The alignment extension terminates when either the end of one of the two sequences is reached or the alignment score decreases by x

Algorithm 6 The matrix computation in ELBA.

```
1: procedure ELBA
2:     reads ← FastaReader()
3:     k-mers ← KmerCounter()
4:     A ← GenerateA(reads, k-mers)                    ▷ Data matrix
5:     A^T ← Transpose(A)
6:     C ← AA^T                                        ▷ Candidate overlap matrix
7:     C ← Apply(C, Alignment())                       ▷ Run alignment
8:     R ← Prune(C, AlignmentScoreLessThan(t))
9:     S ← TransitiveReduction(R)                      ▷ Algorithm 7
10:    return S
11: end procedure
```

compared to the best score obtained up to that point. Each alignment extension of a given read pair can be computed independently.

Data Partitioning

The input is a set of nucleotide sequences in FASTA format [108, 61]. To ensure load balance, each processor reads an equal, independent portion of this file via parallel MPI I/O. Immediately thereafter, the processors begin communicating the sequences to create a 2D grid that matches the way the matrices are partitioned. This approach is similar to the one used by PASTIS [144].

k-mer Selection and Counting

A k-mer based approach calculates the frequency of each k-mer in the input because not all k-mers are useful. k-mers that are usually discarded are (a) k-mers that occur only once in the input (singletons) and (b) high frequency k-mers. For more details on k-mer selection, see the BELLA paper [83].

ELBA eliminates singletons using a Bloom filter [114] during k-mer counting and high frequency k-mers that occur at least d times, as in our first implementation. The threshold d is computed using the approach presented in BELLA [83], which uses dataset-specific features. In Section 3.3, we use d to compute the communication cost of our algorithm.

Our k-mer counter is similar to that of HipMer [70] and consists of two phases. First, we add k-mers to the Bloom filter and then compute the frequencies for the filtered k-mers. The processors extract k-mers from their local sequences, hash them, and possibly communicate them with other processors as specified by the Bloom filter hash function. On the receiver, incoming k-mers are added to the local Bloom filter; if they already exist, they are added to the local hash table partition. Communication requires an all-to-all exchange and is implemented using MPI Alltoall and MPI Alltoallv.

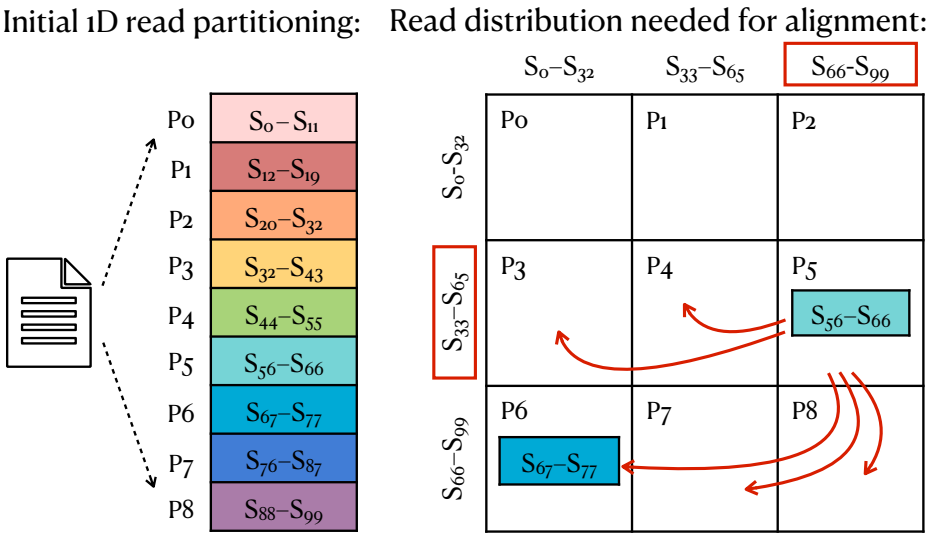

Figure 3.3: Read distribution when we read the input (left) and read distribution we need to perform pairwise alignment (right).

Overlap Detection and Alignment

The local k-mer hash table and the local sequences are used to create a distributed $|sequences|$-by-$|k\text{-}mers|$ matrix $\mathbf{A}$. A nonzero $\mathbf{A}_{ij}$ stores the position of the jth k-mer in the ith sequence. $\mathbf{A}$ is multiplied by $\mathbf{A}^\mathsf{T}$ to obtain the sparse candidate overlap matrix $\mathbf{C} = \mathbf{A}\mathbf{A}^\mathsf{T}$ of dimension $|sequences|$-by-$|sequences|$. In $\mathbf{C}$ each nonzero $\mathbf{C}_{ij}$ stores the number of common k-mers and their positions in the sequence pair i and j. The number of stored positions is a user-defined parameter. For this work we store two k-mer positions for each read pair.

To compute $\mathbf{C}$ we use the distributed SpGEMM in the CombBLAS library [29] and we overload the addition and multiplication operators in SpGEMM with a custom semiring. We overload the multiplication with an assignment by taking the positions of the respective k-mer in two sequences corresponding to $\mathbf{A}_i$ and $\mathbf{A}_j^\mathsf{T}$. We overload the addition operator by incrementing the counter of common k-mers between $\mathbf{A}_i$ and $\mathbf{A}_j^\mathsf{T}$ and storing the positions of another common k-mer in $\mathbf{C}_{ij}$ (i.e., concatenate the results of the multiplication operation) as long as it is smaller than the number of positions to be stored.

CombBLAS relies on 2D Sparse SUMMA algorithm for parallel SpGEMM [24] and it uses a hybrid hash table and heap based algorithm for local multiplication. For matrices, CombBLAS uses a 2D matrix decomposition, so both $\mathbf{A}$ and $\mathbf{C}$ are distributed over a process grid of $\sqrt{P} \times \sqrt{P}$. Observe that in such a distribution a processor may need to align a pair of sequences which it does not have in its partition (Figure 3.3). Such sequences need to be communicated among respective processors. In this respect, a processor has two possibilities: (a) wait until $\mathbf{C}$ is computed to find out which sequences it would need, and

then begin communicating those sequences, or (b) request the full range of sequences it might need once the FASTA input file is read from disk, as described in Section 3.2. We choose the latter option because it allows for overlapping sequence exchanges with k-mer counting and matrix multiplication. This is also the approach adopted by PASTIS [144]. As in PASTIS [144], we implement the second option, which also allows to overlap the exchange of sequences with k-mer counting and matrix multiplication. Figure 3.3 shows how the sequences are distributed when we read the input (on the left) and how they are distributed to perform alignment (on the right). P_5 has local sequences S_{56-66} after reading the input file, however it needs sequences but it needs to fetch sequences from other processors to perform the alignment on sequences assigned to it in the 2D grid. For example, P_5 must get S_{67-77} from P_6. Once the sequences are communicated, we perform pairwise alignment for all identified pairs (i.e., the nonzeros $\mathbf{C}$) using a seed-and-extend algorithm that returns an alignment score and updated seed coordinates. If the score does not exceed the specified threshold, the read pair is discarded and the nonzero is removed from $\mathbf{C}$.

From a matrix point of view: (a) pairwise alignment is an in-place element-wise operation on $\mathbf{C}$ that sets the alignment flag for each nonzero $\mathbf{C}_{ij}$ to true if the alignment score exceeds the given threshold and to false otherwise, and (b) removing entries with false flags is another in-place operation on $\mathbf{C}$ that prunes nonzeros whose flags are set to false. The resulting matrix $\mathbf{R}$ (line 8 in Algorithm 6) is the input to the transitive reduction. Contained sequences, as defined in Chapter 2, are discarded during transitive reduction regardless of their alignment scores. They can be reintroduced at later stages of the *de novo* assembly process.

3.3 Communication Analysis

In this section, we analyze the communication costs for ELBA's k-mer counting, overlap detection, and read exchange, and compare them to our 1D implementation. First, we consider the communication cost of the k-mer counting phase (which is common to both implementations). Then, we examine the communication costs of the overlap phase and the read exchange, which are the main differences between the two implementations. The communication costs are expressed in number of words W (bandwidth cost) and number of messages Y (latency cost). The communication costs are summarized in Table 3.1 along with the useful notations in Table 3.2.

Communication Cost of k-mer Counting

The communication cost for this step depends on the properties of the input dataset and the settings of our algorithm, such as the depth d of the dataset, the genome size G (in nucleotides), the k-mer length k, the number of sequences n in the input, and the sequence length l.

Table 3.1: Communication costs of diBELLA 1D and ELBA.

Task	Bandwidth		Latency	
	diBELLA 1D	ELBA	diBELLA 1D	ELBA
K-mer Counting	$nlk/4P$	$nlk/4P$	bP	bP
Overlap Detection	a^2m/P	$am/\sqrt{P}$	P	$\sqrt{P}$
Read Exchange	cnl/P	$2nl/\sqrt{P}$	$\min\{cnl/P, P\}$	$\sqrt{P}$

Our total input size is $Gd \approx nl$. Each processor has $(1/P)$th of the input. Each sequence has $(l-k+1)$ k-mers and each k-mer requires $k/4$ bytes at 2-bit compression per nucleotide. Therefore, the total size on each processor before communication is $n(l-k+1)k/4P$. For long read data, $l-k+1 \approx l$, since l is usually 2-3 orders of magnitude larger than k.

The hash function distributes k-mers evenly and randomly among processors, so that each processor keeps $(1/P)$th of the data for itself and communicates the rest. For large P, we can assume $(P-1)/P \approx 1$ to avoid clutter. Thus, the bandwidth cost for k-mer counting per process is on average:

$$W = \frac{P-1}{P}\frac{n(l-k+1)k}{4P} \approx \frac{nlk}{4P} \tag{3.11}$$

Depending on the available memory, we may need to perform multiple k-mer exchanges. Thus, the latency cost of k-mer counting is $Y = bP$, where b is the batch count.

Communication Cost of Overlap Detection

In our application, $\mathbf{AA}^\mathsf{T}$ is the output matrix $n \times n$, $\mathbf{A}$ and $\mathbf{A}^\mathsf{T}$ are the input matrices of dimension $n \times m$ and $m \times n$, respectively. The nonzeros in $\mathbf{A}$ and $\mathbf{A}^\mathsf{T}$ are am, where a is the density indicating the average number of sequences containing a given k-mer. The k-mer selection procedure presented in [83], which we use here, chooses an interval for the k-mer frequency, which in turn indicates the average number of sequences that may contain a given k-mer.

Our previous work, diBELLA 1D, computes overlap detection using distributed hash tables [57]. k-mers are distributed among processors that allow them to detect candidate overlap pairs locally. Subsequently, a global reduction must be performed. In terms of communication, this implementation is equivalent to a 1D sparse matrix multiplication using the outer product algorithm [25, 13]. The 1D formulation of the outer product distributes $\mathbf{A}$ in block columns, where the ith block column is denoted by $\mathbf{A}_{:i}$, and $\mathbf{A}^\mathsf{T}$ in block rows, where the ith block row is denoted by $\mathbf{A}^\mathsf{T}_{i:}$. $\mathbf{C}$ is distributed in block rows in diBELLA 1D. The calculation can be written as $C = \sum_{i=1}^{P} \mathbf{A}_{:i}\mathbf{A}^\mathsf{T}_{i:}$.

Each k-mer exists on average in a sequences, so local overlap detection $\mathbf{A}_{:i}\mathbf{A}^\mathsf{T}_{i:}$ generates a^2m/P nonzeros on each processor. These nonzeros must be reduced before performing

Table 3.2: List of symbols and annotations used in this section and in the following one.

Symbol	Description
n	Read set cardinality
m	K-mer set cardinality
d	Depth of coverage
k	K-mer length
L	Overlap length
l	Read length
$\mathbf{A}$	Data matrix: reads–by–kmers
$\mathbf{C}$	Candidate overlap matrix: reads–by–reads
$\mathbf{R}$	Overlap matrix: reads–by–reads
$\mathbf{S}$	String matrix: reads–by–reads
a	$\mathbf{A}$ average density: $^{\text{nnz}(\mathbf{A})}/_{m}$
c	$\mathbf{C}$ average density: $^{\text{nnz}(\mathbf{C})}/_{n}$
r	$\mathbf{R}$ average density: $^{\text{nnz}(\mathbf{R})}/_{n}$
s	$\mathbf{S}$ average density: $^{\text{nnz}(\mathbf{S})}/_{n}$
P	Total number of processes
W	Bandwidth cost
Y	Latency cost

pairwise alignment, so that no read-read pair is aligned more than a few (1-2) times, depending on the parameters of the algorithm. This means that each processor exchanges $W_{1D} = a^2 m/P$ words and the latency cost is $Y_{1D} = P$.

In contrast, ELBA uses a 2D sparse matrix multiplication algorithm known as Sparse SUMMA [24]. The P processors are logically organized in a $\sqrt{P} \times \sqrt{P}$ grid with row and column indices such that the (i, j)th processor is P_{ij}. Each processor stores a $n/\sqrt{P} \times m/\sqrt{P}$ submatrix $\mathbf{A}_{ij}$ and a $m/\sqrt{P} \times n/\sqrt{P}$ submatrix $\mathbf{A}_{ij}^{\mathsf{T}}$ in its local memory. Each processor computes a product of a block row of $\mathbf{A}$ with a block column of $\mathbf{A}^{\mathsf{T}}$. Sparse SUMMA is an owner-computes algorithm, so we only need to consider the communication of the input matrices. If we assume a good load balance, which we achieve by randomly permuting k-mers and reads, $\mathbf{A}_{ij}$ has am/P nonzeros. Each processor P_{ij} receives $2(\sqrt{P} - 1)$ input blocks because $\mathbf{C}_{ij} = \sum_{i=k}^{\sqrt{P}} \mathbf{A}_{ik}\mathbf{A}_{kj}^{\mathsf{T}}$.

For large P, we simplify $\sqrt{P} - 1 \approx \sqrt{P}$ so that the number of nonzeros a processor must collect is $W_{2D} = am/\sqrt{P}$ and the latency cost is $Y_{2D} = \sqrt{P}$.

Communication Cost of Read Exchange

The communication cost of the read exchange follows from the analysis in the previous section. The sequences are distributed by parallel I/O to the processors according to the

Table 3.3: List of experimental values of sparsity for ELBA.

Dataset	**Depth (d)**	**C density (c)**	**Inefficiency ($c/2d$)**	**R density (r)**
E.coli (Sample)	30	145.9	2.4	6.4
C. elegans	40	1,579.7	19.7	8.1
H. sapiens	10	1,207.7	60.4	1.3

corresponding implementation decomposition, i.e., 1D for our first implementation and 2D for the present work.

To compute the communication cost, consider the candidate overlap matrix $\mathbf{C}^{n\times n} = \mathbf{A}\mathbf{A}^{\mathsf{T}}$. $\mathbf{C}$ has cn non-zeros (before computing pairwise alignment), where c is the density per row or column, which is the average number of overlapping sequences for each read. Each exchange costs $O(l)$. The 1D algorithm exchanges at most $W_{1D} = cnl/P$ words and sends $Y_{1D} = \min\{cnl/P, P\}$ messages, while the current 2D implementation exchanges at most $W_{2D} = 2nl/\sqrt{P}$ words and sends $Y_{2D} = \sqrt{P}$ messages.

diBELLA 1D communicates at most one read per nonzero: an alignment is assigned to a processor only if that processor has at least one of the two sequences involved. ELBA communicates the full range of sequences that a processor might need, and starts the read exchange after the initial data partition, so that communication overlaps with computation.

The 1D algorithm scales better with increasing parallelism, but has a large constant c whose typical value often exceeds 1000 for large genomes, as shown in Table 3.3. To overcome this large constant and communicate fewer words than the 2D algorithm, the 1D algorithm would require $(c^2/4)$-way parallelism. Ellis et al. [57] show that $c \approx 2d$ for a perfect overlapper. In practice, c is much larger than d and $c/2d$ can be considered the *inefficiency factor* of an overlapper.

3.4 Experimental Setup

Our runtime evaluation was performed on two machines: the Cray XC40 supercomputer Cori at NERSC and the IBM supercomputer Summit at Oak Ridge National Laboratory (Table 3.5). On Cori we use the Haswell partition, while on Summit we use only IBM POWER9 CPUs. The use of two architectures shows that our algorithm scales on different architectures. However, this is not intended as a cross-platform comparison, as our algorithm is not optimized specifically for either platform, but for a general HPC architecture.

Each Haswell node on the Cori system consists of two 2.3GHz 16-core Intel Xeon processors and has a total memory of 128GB. Each Summit node has two 22-core IBM POWER9 processors and 512 GB of DDR4 from RAM. Because one core per Summit half-node is reserved for OS, each node has a maximum of 42 cores available for application codes. In this dissertation, we do not use the GPUs available on Summit.

Table 3.4: Datasets used for accuracy assestment. Datasets above the line are real data, while datasets below the line were generated using PBSIM [128]. Download: `portal.nersc.gov/project/m1982/bella/`.

Dataset	Depth (d)	Species and Strain	Fastq Size (MB)
E.coli (Sample)	30	*Escherichia coli* MG1655	266
E.coli	100	*Escherichia coli* MG1655	929
E. coli (CCS Sample)	29	*Escherichia coli* MG1655	240
E. coli (CCS)	290	*Escherichia coli* MG1655	2,600
C. elegans	40	*Caenorhabditis elegans* Bristol	8,900
P. aeruginosa	30	*Pseudomonas aeruginosa* PAO1	359
V. vulnificus	30	*Vibrio vulnificus* YJ016	288
A. baumannii	30	*Acinetobacter baumannii*	248
C. elegans	20	*Caenorhabditis elegans*	3,750

Table 3.5: Details of the machines used for evaluation: name, number of physical cores per node, maximum processor turbo frequency, processor model, memory, network, and size of L1, L2, and L3 caches.

Platform	Cores/Node	Freq (GHz)	Processor	Memory (GB)	Network	L1	L2	L3
Cori Haswell	32	3.6	Intel Xeon E5-2698V3	128	Aries Dragonfly	64KB	256KB	40MB
Summit CPU	42	4.0	IBM POWER9	512	InfiniBand Fat Tree	32KB	512KB	10MB

Table 3.6: Genomes used in the runtime performance evaluation: name, depth, number of sequences in the input, average read length, input size, genome size, and error rate.

Dataset	Depth (d)	Reads (K)	Length	Input (GB)	Size (Mb)	Error Rate
C. elegans	40	420.7	11,241	4.8	100	0.13
H. sapiens	10	4,421.6	7,401	33.1	3,000	0.15

To investigate the parallel performance of our algorithm, we use two genomes sequenced with Pacific Biosciences (CLR technology) [140] with different sizes: Caenorhabditis elegans (C. elegans) and Homo sapiens (H. sapiens). Details of the two datasets can be found in Table 3.6. Our algorithm is also suitable for other long-read technologies such as Pacific Biosciences CCS [158] (or HiFi) and Oxford Nanopore [96].

The experiments are divided into two groups: (a) parallel performance and scalability of ELBA and (b) performance comparison with related work. In the latter, we compare the overlap detection of ELBA with diBELLA 1D [57], which is also implemented for distributed memory parallelism, and minimap2 [104], which is written in C for shared memory. minimap2 is one of the most popular and fastest shared memory algorithms for overlap detection. For

this reason, we decided to report its runtime compared to ELBA.

ELBA and diBELLA 1D ran with the same input and alignment setting, i.e. $k = 17$ and a maximum k-mer frequency of 4, while minimap2 ran with its default setting for CLR data. The minimap2 results were collected only from Cori, since minimap2 uses SSE intrinsics, which are not supported on the IBM POWER9 processor. For minimap2, we report only single node performance since it does not implement distributed memory parallelism.

On Cori, we used `gcc-8.3.0` and the `O3` flag to compile C/C++ codes, while on Summit we used `gcc-8.1.1`. On both Cori and Summit, we used the default MPI implementation. Here, we report the average runtime over 10 runs for each experiment, except for the H. sapiens dataset at low concurrency, where we report the average over three runs.

Genomes used for accuracy assessment are listed in Table 3.4. The selected genomes vary in size and complexity, as these features affect accuracy [105]. In addition, we include a dataset based on Pacific Biosciences' CCS technology, which has a significantly lower error rate than Pacific Biosciences' CLR technology sampled at different depth. The CCS data is more accurate than the CLR data, but has a higher cost and a slightly shorter read length (although it is still classified as a long read technology). The synthetic data were generated using PBSIM [128] with an error rate of 15%. Recall, precision, and F1 score are used as accuracy metrics. Recall is defined as the proportion of true positives from the aligner/overlapper to the total ground truth. Precision is the proportion of true positives from the aligner/overlapper to the total number of elements found by the aligner/overlapper. The F1 score is the harmonic mean of precision and recall.

A read pair is considered true-positive if the sequences align for at least 2 kb in the reference genome. The threshold $t = 2$ kb was derived from the procedure proposed by Heng Li [103] and the ground truth was generated using minimap2.

3.5 Results

Figure 3.4 illustrates the strong scaling of our algorithm for C. elegans on the left and for H. sapiens on the right. The two machines run on $P = \{32, 72, 128\}$ nodes using 32 MPI ranks/nodes for C. elegans and $P = \{128, 200, 288, 338\}$ for H. sapiens. For C. elegans, ELBA achieves a parallel efficiency of 83% on Summit CPU and of 68% on Cori Haswell. For H. sapiens, the parallel efficiency of both machines is over 80% with a peak of 92% on Summit CPU. These results demonstrate the near linear scaling behavior of our algorithm with a large input on two different architectures.

Figures 4.1-4.2 (C. elegans) and Figures 4.3-4.4 (H. sapiens) show the runtime breakdown of ELBA on the two machines. Plots on the left side of the figures show the total execution time including pairwise alignment, while plots on the right side exclude pairwise alignment. We included plots without the pairwise alignment because the alignment takes up a large portion of the runtime and makes it difficult to see the scaling of the other stages. From bottom to top, the layers are ordered according to the legend. The first layer is the pairwise alignment, i.e., the time required to align all non-zero pairs in the candidate overlap matrix

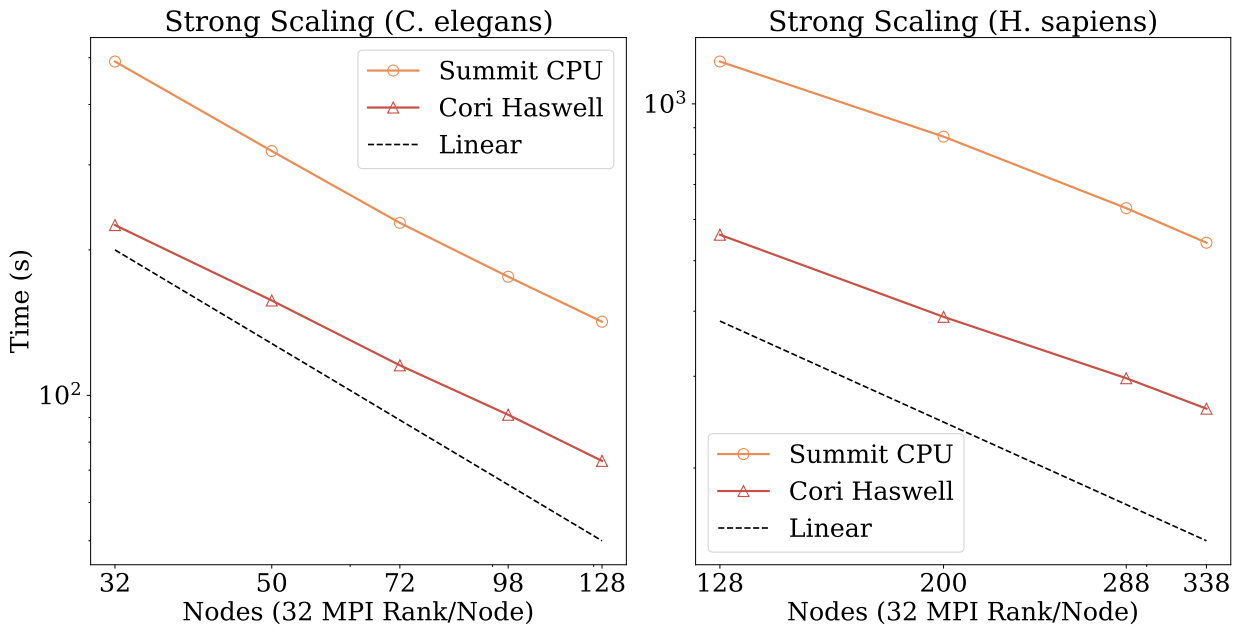

Figure 3.4: ELBA strong scaling on Cori Haswell and Summit CPU using 32 MPI rank/node on C. elegans (left) and on H. sapiens (right).

C. `ReadFastq` is the time taken to read and parse the input file in parallel. Immediately after this step, we start exchanging sequences to overlap this communication with the subsequent computation. `CountKmer` corresponds to the k-mer counting step. `DetectOverlap` includes both the communication time and the computation time to create the candidate overlap matrix $\mathbf{C} = \mathbf{A}\mathbf{A}^\mathsf{T}$. `ReadExchange` is the time from the end of `SpGEMM` to the completion of all sequence exchanges. Depending on the MPI implementation, the read exchange may overlap with k-mer counting and overlap detection phases.

Figure **??** shows the breakdown of ELBA performance on the C. elegans dataset using P={32, 72, 128} nodes on Summit CPU. Cori Haswell has similar scaling for each stage so we only report the breakdown for Summit CPU. In general, ELBA runs faster overall on Cori Haswell than on Summit CPU. The relative contribution of pairwise alignment to the overall runtime increases on Summit CPU compared to Cori Haswell. SeqAn's pairwise alignment is probably not optimized for IBM processors, so we refrain from making architectural comparisons based solely on this data. Overlap detection ($\mathbf{A}\mathbf{A}^\mathsf{T}$) has the largest contribution to runtime after pairwise alignment. Parallel read I/O shows no scaling and its performance degrades as the number of processes is increased. For C. elegans, overlap detection has a parallel efficiency of 63% for Summit CPU, while k-mer counting (consisting of first and second passes) has a parallel efficiency of 80%. The read exchange parallel efficiency is 50%. Figure 3.6 shows a similar story for H. sapiens. The size of the dataset, which is about 10 *times* that of C. elegans, mitigates the contribution of I/O to runtime. The parallel

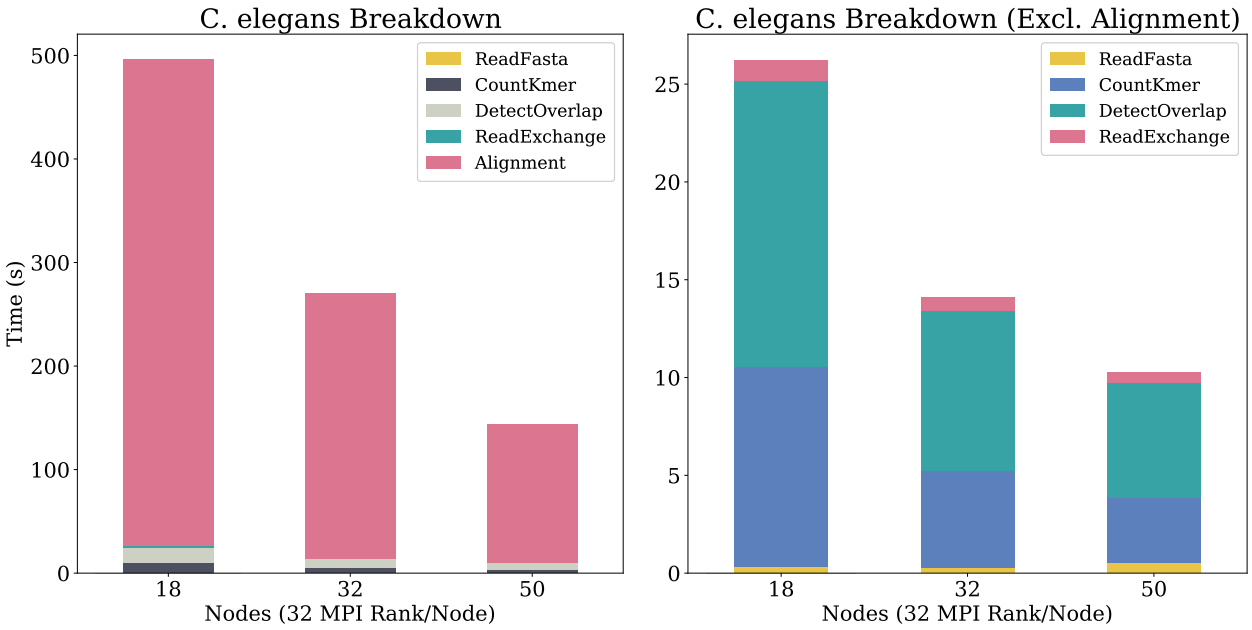

Figure 3.5: ELBA runtime breakdown on Summit CPU (C. elegans).

efficiency of $\mathbf{A}\mathbf{A}^\mathsf{T}$ increases to 65%. The formation of $\mathbf{A}$ and $\mathbf{A}^\mathsf{T}$ has a negligible impact on runtime, but scales almost linearly with a parallel efficiency above 80%.

First, we compare the overall runtime and scaling of ELBA with diBELLA 1D [57]. This comparison was performed on Summit CPU and is shown in Figure 3.7 for C. elegans and H. sapiens. ELBA and diBELLA 1D differ mainly in the way they perform overlap detection and communicate sequences before pairwise alignment. They show similar near-linear scaling behavior, but ELBA consistently outperforms diBELLA 1D by 1.5-1.9× for C. elegans and 1.2-1.3× for H. sapiens. For completeness, we evaluate ELBA against minimap2 [104], a popular shared memory algorithm for overlap detection. Thus, we run minimap2 on a single node with 32 OpenMP threads and compare its runtime against ELBA on a different set of nodes with 32 MPI ranks/nodes. It is important to note that minimap2 and ELBA perform significantly different computations. In particular, minimap2 does not perform base- level pairwise alignment and instead estimates pairwise similarity based on the number of shared minimizers, making it significantly faster. Nevertheless, they ultimately aim to solve the same problem, which is why we make a comparison here. For the running times of ELBA, we refer to Figure 3.4. For C. elegans, minimap2 is about 2× faster than ELBA at P = 8, while ELBA is 1.6×, 3.2×, and 5× faster than minimap2 at higher concurrency. The speedup of ELBA over minimap2 is 9.5×, 13.7×, and 20.6× at P=$\{128, 200, 338\}$ for the H. sapiens dataset.

ELBA is evaluated against several state-of-the-art long read overlap detection and alignment software packages summarized in Chapter 7, using both synthetic and real data from

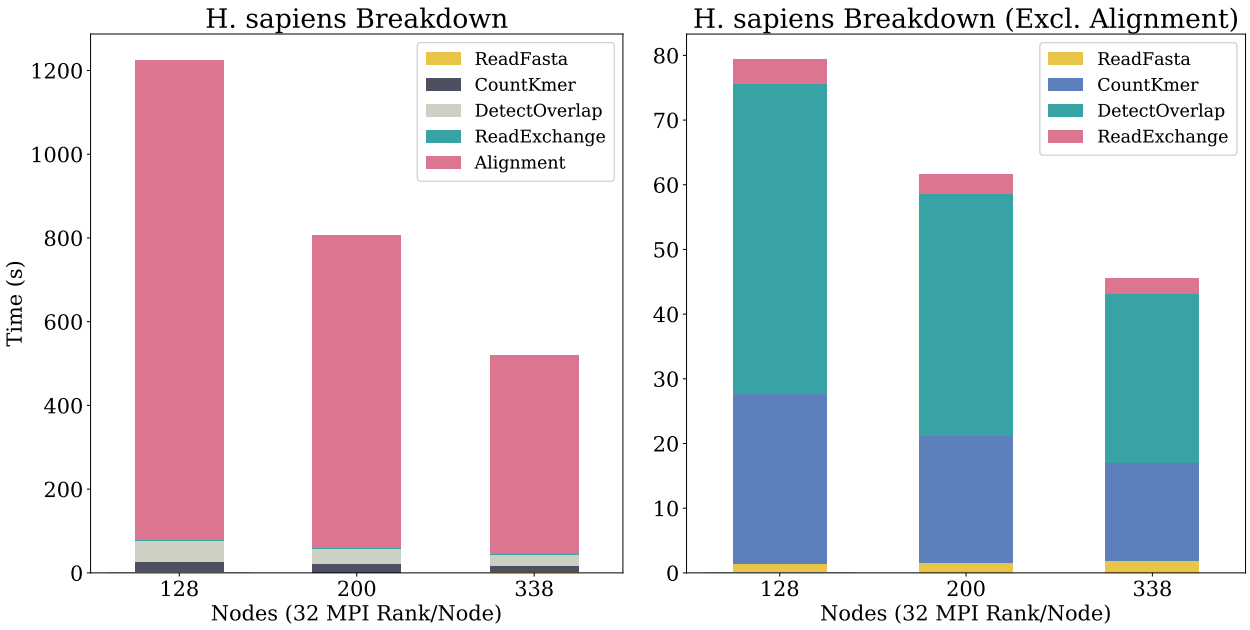

Figure 3.6: ELBA runtime breakdown on Summit CPU (H. sapiens).

Pacific Biosciences (Table 3.4). The advantage of synthetic data is that the ground truth is known. Table 3.7 and Table 3.8 show the accuracy for synthetic and real data, respectively. The last column of each table indicates whether the respective overlapper also computes alignment.

Table 3.7 shows that MECAT trades recall for precision, achieving the highest precision but overlooking a large number of the true overlaps. In contrast, ELBA, minimap2, and BLASR were consistently strong (generally above 80%) in both precision and recall. The F1 score of ELBA is consistently higher than that of the competing software, with the exception of minimap2, which had a slight improvement of 1.1% in three of four genomes, while ELBA had an improvement of 1.2% over Miniamp2 in C. elegans 20X. Table 3.8 shows that while BLASR performed reasonably well on synthetic data, it had a lower hit rate than other software on real data. The F1 score of ELBA outperformed competing software except minimap2 on E. coli 100X. It is worth noting that the precision and F1 score of ELBA are significantly better than competing software for CCS data, the sequencing data with lower error rates.

Tables 3.7 and 3.8 show the competitive accuracy of ELBA compared to the literature and demonstrate the effectiveness of the methodologies we have introduced and implemented. For synthetic data, ELBA achieves both high recall and precision, consistently among the best. For real CLR data, ELBA's recall and precision are generally lower than for synthetic data, yet ELBA's F1 results are among the best and show performance stability that competing software often does not. Notably, ELBA has a 49.16% higher F1 score than minimap2 for C.

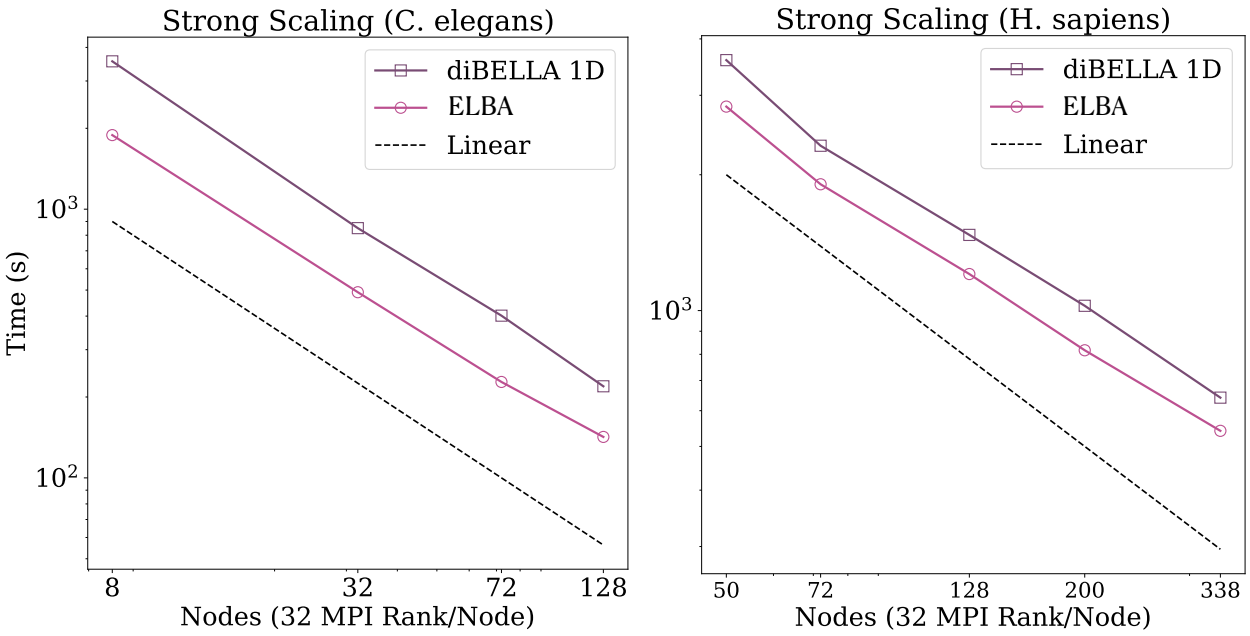

Figure 3.7: Comparison of ELBA and diBELLA 1D [57] on Summit CPU.

elegans 40X. The precision and F1 score of ELBA on real CCS data are significantly better than those of the competing software. Overall, a good score on one dataset becomes one of the worst on another, while ELBA's F1 score is within 1.7% of the top performer.

Tables 3.7 and 3.8 show that ELBA achieves higher F1 score values on synthetic data and real CCS data than on real CLR data. The way ground truth is generated could explain this behavior. For synthetic data, the ground truth comes directly from the data set itself. So we know exactly where a read in the reference genome comes from and which other sequences overlap with it. For real data, the positions of sequences in the reference are determined by mapping the sequences to the reference using minimap2 in "mapping mode." Intuitively, such a procedure is suboptimal, since there is no guarantee that minimap2 will correctly locate each read. ELBA could potentially find a better set of true overlaps than those identified by minimap2. Given a uniformly covered genome, we observed that minimap2 and other long-read mappers tend to map sequences to "hotspots" within a genome rather than mapping them uniformly across the genome. This leads to uneven coverage and overestimation of overlaps by a factor of 1.14×. Recall beyond a certain point, for real data, would mean that the overlapper also overestimates the overlap cardinality. It is possible that the actual accuracy of ELBA is actually higher for real data. As future work, we plan to investigate these issues in more detail. This bias may not be present when CCS data are mapped to the reference, as error rates are lower and it is easier for the mapper to find the correct position on the reference genome.

Table 3.7: Recall, precision, and F1 score for synthetic data. The last column indicates whether the tool computes a nucleotide-level alignment. **Bold font** indicates best performance, underlined font indicates second best. DALIGNER was not run on synthetic data due to runtime error.

Dataset	Overlapper	Recall	Precision	F1 Score	Pairwise Alignment
P. aeruginosa 30X	ELBA	97.66	89.68	93.50	Y
	BLASR	86.86	90.54	88.66	Y
	MECAT	38.40	**95.20**	54.72	N
	Minimap2	**99.10**	88.83	**93.69**	N
	MHAP	72.68	63.42	67.74	N
V. vulnificus 30X	ELBA	**97.27**	84.05	90.18	Y
	BLASR	87.31	84.74	86.01	Y
	MECAT	43.53	88.89	58.44	N
	Minimap2	96.71	**89.33**	**92.87**	N
	MHAP	74.52	45.10	56.19	N
A. baumannii 30X	ELBA	**97.54**	84.90	90.78	Y
	BLASR	89.51	84.58	86.98	Y
	MECAT	46.31	**90.39**	61.25	N
	Minimap2	96.89	85.79	**91.01**	N
	MHAP	76.88	28.79	41.89	N
C. elegans 20X	ELBA	91.80	88.02	**89.87**	Y
	BLASR	95.61	78.19	86.02	Y
	MECAT	13.45	**95.09**	23.57	N
	Minimap2	**95.76**	82.84	88.83	N
	MHAP	82.57	6.41	11.90	N

3.6 Summary

Long-read sequencing technologies enable highly accurate reconstruction of complex genomes. Read overlap is a major computational bottleneck in long-read pipelines for genome analysis, such as genome assembly.

In this section, we introduced the overlap detection and alignment phases of ELBA. ELBA uses a k-mer-based approach for overlap detection in long-read sequencing data. Then we demonstrated the feasibility of the k-mer-based approach using a mathematical model based on Markov chains and showed the generality of such a model. ELBA provides a novel algorithm for pruning k-mers that are unlikely to be useful for overlap detection and whose presence would only add unnecessary computational cost. Our algorithm for reliably detecting k-mers explicitly maximizes the probability of retaining k-mers that belong to unique regions of the genome.

Table 3.8: Recall, precision, and F1 score for real data. BLASR result for C. elegans 40X is not reported as BLASR `v5.1` does not accept `fastq` larger than 4 GB.

Dataset	Overlapper	Recall	Precision	F1 Score	Pairwise Alignment
E. coli (Sample)	ELBA	82.66	**85.69**	**84.15**	Y
	DALIGNER	89.97	62.62	73.84	Y
	BLASR	77.64	79.64	78.63	Y
	MECAT	78.41	78.71	78.56	N
	Minimap2	**91.40**	76.36	83.21	N
	MHAP	79.71	66.93	72.76	N
E. coli	ELBA	65.08	71.22	68.01	Y
	DALIGNER	82.18	54.50	65.54	Y
	BLASR	35.41	72.01	47.48	Y
	MECAT	54.61	**72.69**	62.37	N
	Minimap2	**80.68**	62.30	**70.30**	N
	MHAP	67.84	44.60	53.81	N
E. coli (CCS Sample)	ELBA	96.32	**99.84**	**98.05**	Y
	BLASR	92.38	97.30	94.77	Y
	MECAT	98.21	88.39	93.04	N
	Minimap2	98.90	58.34	73.39	N
	MHAP	**99.05**	38.29	55.23	N
E. coli (CCS)	ELBA	97.67	99.81	**98.73**	Y
	BLASR	9.11	**100.00**	16.70	Y
	MECAT	15.71	99.95	27.15	N
	Minimap2	98.83	69.94	81.91	N
	MHAP	**98.99**	38.80	55.75	N
C. elegans 40X	ELBA	75.43	73.81	**74.61**	Y
	DALIGNER	62.81	58.66	60.67	Y
	MECAT	73.05	**75.27**	74.14	N
	Minimap2	**94.13**	34.06	50.02	N
	MHAP	86.63	5.31	10.01	N

ELBA achieves rapid overlap without sketching using sparse matrix multiplication (SpGEMM) implemented with high-performance software and libraries developed for this subroutine. Any novel sparse matrix format and multiplication algorithm would be applicable to overlap detection and would enable further performance improvements. Moreover, our SpGEMM approach is general and flexible enough that it can be coupled with any k-mer selection strategy. Our overlap detection is coupled with a seed-and-extend alignment algorithm to filter out spurious overlap candidates. In this context, we developed a new method to separate true overlap candidates from false positives as a function of alignment score. This method shows that the probability of false positives decreases exponentially as the overlap

length between sequences increases.

ELBA consistently achieves high overlap detection accuracy compared to state-of-the-art software, both for synthetic and real data. Our communication analysis shows that the new two-dimensional SpGEMM-based overlap detection algorithm reduces communication compared to the existing 1D algorithm based on distributed hash tables for commonly used concurrences in the range of 100-10000 processes. This translates to a speedup of 1.2-1.9× in our experiments. Furthermore, for C. elegans, ELBA is 1.6×, 3.26×, and 56× faster than minimap2 at high concurrences while the speedup of ELBA over minimap2 is 9.56×, 13.7×, and 20.6×, at $P = \{128, 200, 338\}$ for H. sapiens.

Chapter 4

Parallel Algorithms for Transitive Reduction

In this chapter we describe the foundation of transitive reduction from the point of view of a graph, and then describe our parallel algorithms for transitive reduction using sparse matrices and semiring abstraction.

4.1 Overview and Foundation

In Chapter 3 we gave an overview of the Overlap step of the OLC assembly paradigm and described our approach to the computational challenge. In this chapter, we turn to the second step of the OLC paradigm: Layout. The goal of this step is to simplify the overlap graph (or for us, the overlap matrix) into a *string graph* (or string matrix) so that we remove redundant information due to redundant sequencing and inherent genome repetitiveness. A string graph has the desirable property of combining genomic repetitions into a single unit [148]. This transformation facilitates the clustering of the graph and hence the generation of the *contig* step, which will be the subject of the next chapter. A contig is a set of overlapping sequences that together form a consensus region of DNA, where a consensus sequence (or canonical sequence) is the calculated order of the most frequent nucleotides found at each position in a sequence alignment. The formal name for this simplification process is *transitive reduction*.

In graph theory, a transitive reduction of a directed graph D introduced in 1972 by Aho, Garey, and Ullman [1] is another directed graph with the same vertices and as few edges as possible, such that for all pairs of vertices v, w there exists a (directed) path from v to w in D if and only if such a path exists in the reduction. The transitive reduction can be defined for an abstract binary relation on a set by interpreting the pairs of the relation as arcs in a directed graph.

Eugene Myers [119] first presented an algorithm for transitive reduction in linear expected time in the context of *de novo* genome assembly in 2005, but it is sequential in nature. Myers' transitive reduction algorithm consists of iterating over each node v in the source graph and examining nodes that are up to two edges away from v to identify all transitive edges that

exit or enter v [119]. These edges are then marked for removal, and they are removed after all nodes have been considered. Myers' algorithm for transitive reduction is implemented in popular state-of-the-art software for *de novo* genome assembly, such as miniasm [103].

ELBA proposes a new approach that relies on sparse linear algebra to transitively reduce the overlap graph into a string graph [85]. Unlike Myers' algorithm, our transitive reduction algorithm is highly parallel, though not necessarily in linear time. Both the overlap graph and the string graph are represented as sparse matrices, and the entire transitive reduction algorithm is expressed as operations on sparse matrices. Our contributions also include the design of custom semirings, which are essential for the correctness of the algorithm.

4.2 Proposed Algorithm

Our distributed memory transitive reduction algorithm takes the overlap matrix $\mathbf{R}$ as input and computes a transitive reduced version of $\mathbf{R}$, which we denote as $\mathbf{S}$ (line 9 in Algorithm 6). Recall that each $\mathbf{R}_{ij}$ that is not zero stores the number of common k-mers and their positions in the sequence pair (s_i, s_j). The transitive reduction algorithm requires two extra pieces of information for each such pair: the length of the overlap suffix and the overlap orientation. Both can be derived in-place from the alignment coordinates stored in $\mathbf{R}_{ij}$.

Our transitive reduction algorithm is shown in Algorithm 7. If the overlapper returns directed graphs in which an edge (j, i) is missing even though an edge (i, j) is present, the input matrix $\mathbf{R}$ must first be symmetrized. The overlap matrix $\mathbf{R}$ must be structurally symmetric, i.e., if R_{ij} is nonzero, $\mathbf{R}_{ji}$ must also be nonzero, but the nonzero values stored in $\mathbf{R}_{ij}$ and $\mathbf{R}_{ji}$ may be different. The structural symmetry is required because the graph is bidirectional and the edges have four possible orientations (since the DNA consists of two opposite and complementary sequences). The algorithm begins by determining the two-hop neighbors of each vertex in the overlap graph. This first step is the most computationally intensive phase of the transitive reduction and is accomplished by multiplying the overlap matrix $\mathbf{R}$ by itself in the first iteration and by its power in subsequent ones: $\mathbf{N} = \mathbf{P} \cdot \mathbf{R}$, where $\mathbf{N}$ is the two-hop *neighbor* matrix. In *de novo* assembly, if there are multiple alternative paths in the graph, we keep the one that gives us greater genomic coverage in terms of nucleotides. Thus, if there are multiple alternatives, the path with the shorter suffix is chosen, since a shorter suffix indicates longer overlap between two sequences. This is achieved by using a custom MinPlus semiring during the SpGEMM $\mathbf{N} = \mathbf{P} \cdot \mathbf{R}$. Algorithm 8 illustrates the MinPlus semiring we use, where we overload the addition operation with a minimum operation and the multiplication operation with a summation.

Given the bidirectionality of our graph, we ensure that the orientation of the edges in play conforms to the transitivity rules listed in Chapter 2. This is ensured by checking if the edges follow the transitivity rules during multiplication (line 5 in Algorithm 8). If not, we mark the edge as *not transitive*. In particular, we check in MinPlus semiring whether the two heads adjacent to the intermediate node (i.e., the middle node of a path with three

Algorithm 7 Parallel transitive reduction on $\mathbf{R}$.

```
1:  procedure TransitiveReduction(R)
2:      if R ≠ R^T then
3:          R ← R + R^T                    ▷ R must be a structurally symmetric matrix
4:      end if
5:      P ← R                                                  ▷ P is a copy of R
6:      T ← Zero(R)                          ▷ T is a boolean matrix initialized to 0
7:      F ← Apply(R, f) ▷ F is a copy of R with a constant f added to each nonzero R_{i,j}
8:      do
9:          prev ← T.nnz
10:         N ← P · R                                  ▷ Find edges two-hop away
11:         P ← N
12:         B ← F ≥ N                    ▷ B stores transitive edges for this iteration
13:         if B ≠ B^T then
14:             B ← B + B^T              ▷ B must be a structurally symmetric matrix
15:         end if
16:         T ← T + B                          ▷ T accumulates the transitive edges
17:         nnz ← T.nnz
18:     while nnz ≠ prev
19:     S ← R ∘ ¬T                                     ▷ Remove transitive edges
20:     return S
21: end procedure
```

nodes) have opposite directions. For example, a forward overlap and an inner overlap are fine, while a forward overlap and an outer overlap mark the two-hop edge as non-transitive.

Let us assume that this is our overlap matrix $\mathbf{R}$, where zeros are actually zeros in the matrix (i.e., nonexistent entries) and nonzeros represent the overlap suffix. To keep the running example simple, we also assume that the edges exist only in the forward direction and therefore we can consider only the lower triangular part of $\mathbf{R}$:

$$\mathbf{R} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 30 & 0 & 0 & 0 \\ 80 & 30 & 0 & 0 \\ 0 & 80 & 30 & 0 \end{bmatrix}$$

The operation in lines 2—4 of Algorithm 7 ensures that $\mathbf{R}$ is structurally symmetric by using a custom semiring that ensures that a missing zero is not only introduced, but also has the correct value. In line 5, we create a copy of $\mathbf{R}$, which we call a power matrix $\mathbf{P}$, since $\mathbf{P}$ is replaced at each iteration by the neighbor matrix $\mathbf{N}$ from SpGEMM. In line 6, we create a Boolean copy of $\mathbf{R}$, which we refer to as the transitive matrix $\mathbf{T}$, since we store in this matrix the edges that would have to be removed because they are redundant. Any

Algorithm 8 Custom MinPlus semiring used in $\mathbf{N} \leftarrow \mathbf{R}^2$.

```
1: struct MinPlusSR
2:     Id( ) return ∞
3:     Add(a, b) return Min(a, b)                ▷ Find the shortest path
4:     Multiply(a, b)
5:         if IsDirOK( ) then return a + b
6:         else return Id( )
7:         end if
8: end struct
```

nonzero in $\mathbf{T}$ is initialized to $false$ or 0. In line 7 of Algorithm 7, we create the *fuzz* matrix $\mathbf{F}$, in which the overlap length of each nonzero is increased by a constant f to account for sequencing mistakes. In *de novo* assembly, the overlap candidates are approximate matches, since sequencing mistakes can lead to a shift in endpoint positions. To make our algorithm robust to sequencing errors, we increase the value of the longest overlap per row (i.e., per read) by one scalar f. In this example, let us assume $f = 10$:

$$\mathbf{F} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 40 & 0 & 0 & 0 \\ 90 & 40 & 0 & 0 \\ 0 & 90 & 40 & 0 \end{bmatrix}$$

In $\mathbf{R}$ we can go from the first to the second read, from the second to the third, and from the third to the fourth with overlap suffixes of length 30. But we can also go from the first to the third with a suffix of length 80, and similarly from the second to the fourth. Our goal is to obtain the path with the shortest edges, so we want to remove the two edges of length 80 from the original overlap matrix $\mathbf{R}$ (line 10, Algorithm 7).

$$\mathbf{N} = \mathbf{P} \cdot \mathbf{R} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 60 & 0 & 0 & 0 \\ 30 & 60 & 0 & 0 \end{bmatrix}$$

In $\mathbf{N}$ we determine the shortest path from a given read to its two hop neighbors. The shortest path from the first to the third read has length 60, since it passes over two edges of length 30. The path from the first to the third read in a single hop would cost 80, and we want to minimize, not maximize, this distance. The first iteration corresponds to squaring the overlap matrix, because $mP = \mathbf{R}$. Then, in line 11, $\mathbf{P}$ takes the values of $\mathbf{N}$, so in the second iteration $\mathbf{R}$ is multiplied by its square $\mathbf{N} = \mathbf{R} \cdot \mathbf{R}$ to discover three-hop neighbors, and similarly in subsequent iterations. In lines 13–15, we ensure that structural symmetry is maintained with a procedure similar to the one used to make $\mathbf{R}$ symmetrical at the beginning.

The next step is to identify the transitive edges in $\mathbf{N}$ (line 12, Algorithm 7). Our algorithm performs an element-wise operation between $\mathbf{F}$ and $\mathbf{N}$ to identify such edges. If $\mathbf{F}_{ij}$ is greater than or equal to $\mathbf{N}_{ij}$, the corresponding nonzero $\mathbf{B}_{ij}$ in the output matrix is set to *true* or 1. In $\mathbf{N}$, we store the shortest path such that all nonzeros with $\mathbf{F}_{ij} \geq \mathbf{N}_{ij}$ are transitive edges because $\mathbf{F}_{ij}$ is an overlap suffix longer than $\mathbf{N}_{ij}$.

In our running example the output matrix is $\mathbf{B}$:

$$\mathbf{B} = \mathbf{F} \geq \mathbf{N} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

The element-wise operation described is performed only for entries that are nonzero in both $\mathbf{F}$ and $\mathbf{N}$. Recall that in computing $\mathbf{N} = \mathbf{P} \cdot \mathbf{R}$ we checked whether a path is a valid path or not. In this element-wise operation, we also make sure that the orientation of the edges at the intersection of $\mathbf{N}$ and $\mathbf{F}$ follows the last two transitivity rules. In the element-wise operation, we check that the two heads adjacent to the *departure* node (i.e., the start node of a path with three nodes) and the two heads adjacent to the *destination* node (i.e., the end node of a path with three nodes) have the same orientation.

In line 16, we accumulate the edges marked as transitive in this iteration (and stored in $\mathbf{B}$) into $\mathbf{T}$, which stores the entire set of transitive edges found during the computation. If the number of nonzeros in $\mathbf{T}$ does not change between one iteration and the next, we exit the loop. The final operation of our transitive reduction algorithm is to prune the identified transitive edges from $\mathbf{R}$ (line 19, Algorithm 7). This is achieved by an element-wise multiplication of $\mathbf{R}$ and the logical negation of $\mathbf{B}$, $\neg\mathbf{B}$. Any nonzero in $\mathbf{B}_{ij}$ becomes a zero in $\neg\mathbf{B}_{ij}$, so the nonzeros of $\mathbf{R}$ corresponding to transitive edges (i.e., zeros) in $\neg\mathbf{B}$ are pruned. Again, only those entries that are nonzero in both $\mathbf{R}$ and $\neg\mathbf{B}$ are considered. This is equivalent to a set difference operator ($nonzeros(\mathbf{R}) \setminus nonzeros(\mathbf{B})$) in linear algebra.

In our example, $\mathbf{R}$ becomes:

$$\mathbf{S} = \mathbf{R} \circ \neg\mathbf{B} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 30 & 0 & 0 & 0 \\ 0 & 30 & 0 & 0 \\ 0 & 0 & 30 & 0 \end{bmatrix}$$

In this example, all transitive edges are removed after only one iteration of the algorithm, giving us $\mathbf{S}$. In practice, we need several rounds to remove all transitive edges, since we need to consider neighbors that are three, four, etc. hops away. Thus, our algorithm iterates over $\mathbf{R}$ until the number of transitive nonzeros in $\mathbf{T}$ remains the same (line 18, Algorithm 7).

Table 4.1: Communication costs of the transitive reduction phase of ELBA.

Task	**Bandwidth**	**Latency**
Transitive Reduction	$rn/\sqrt{P}$	$t\sqrt{P}$

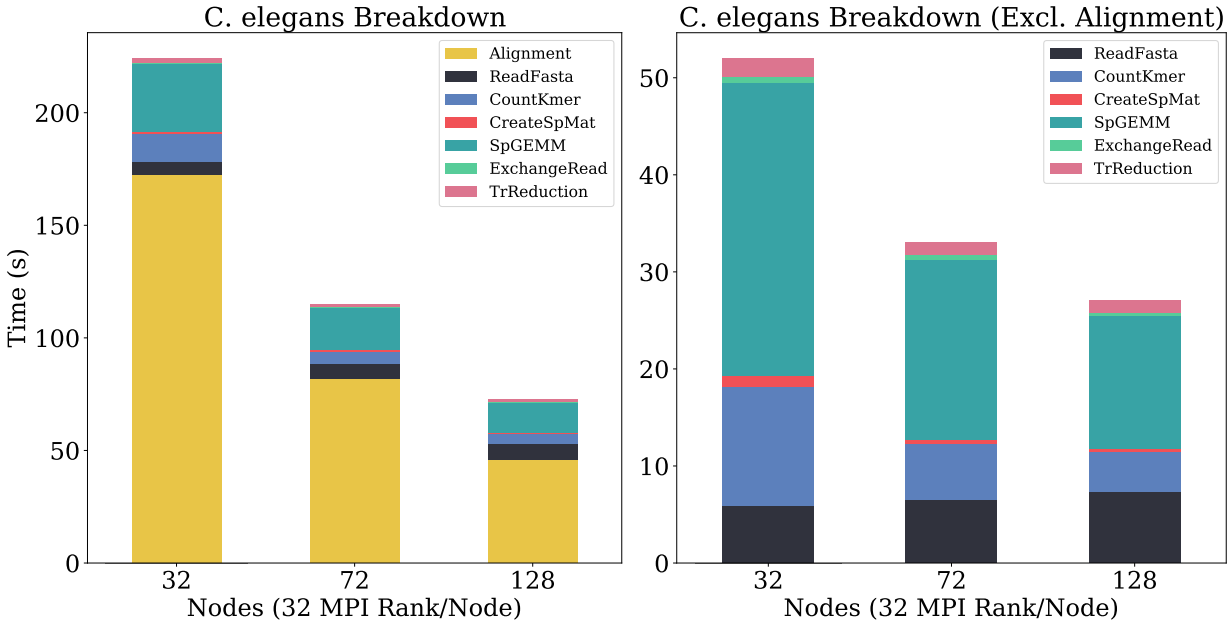

Figure 4.1: ELBA runtime breakdown on Cori Haswell (C. elegans).

4.3 Communication Analysis

In this section we analyze the communication cost for the transitive reduction of ELBA. The SpGEMM dominates the runtime of the transitive reduction algorithm. The communication cost for multiplying $\mathbf{R}$ by its powers follows from the previous analysis and is $W_{2D} = rn/\sqrt{P}$, where $r \leq c$ is the sparsity of the overlap matrix $\mathbf{R}^{n\times n}$ after performing pairwise alignment, which often leads to discarding nonzeros, and $Y_{2D} = \sqrt{P}$. Our transitive reduction also contains some element-wise sparse routines, but these are executed in place, so they do not contribute to communication time. The transitive reduction loop is repeated until convergence, but the number of iterations is often a small constant (denoted t in Table 4.1) and the geometrically decreasing density after each iteration makes the total communication volume asymptotically equal to that of the first iteration. For a summary of the variables, please see Table 3.2.

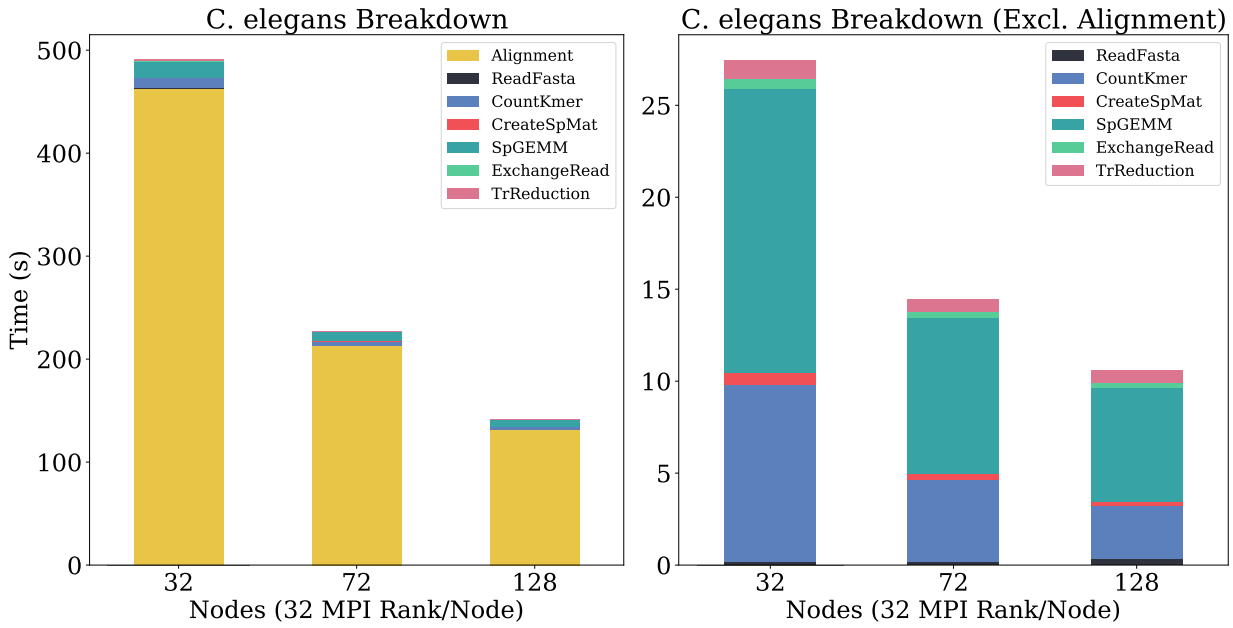

Figure 4.2: ELBA runtime breakdown on Summit CPU (C. elegans).

4.4 Experimental Setup

In terms of hardware, the experimental setup used to evaluate the performance of the transitive reduction phase is the same as that used to evaluate the overlap detection phase in Table 3.5 in Chapter 3. Also, the genomes for this evaluation are the same as those used in Chapter 3.

To evaluate the performance of our transitive reduction, we compare our transitive reduction algorithm with SORA. SORA computes only transitive reduction, so we first generate the overlap graph (or matrix) $\mathbf{R}$ with ELBA and then use it as input to the transitive reduction algorithm of ELBA and SORA. This is an overlap graph consisting of 5.8 million edges and 4.4 million vertices for the H. sapiens dataset and 4.2 million edges and 0.4 million vertices for C. elegans. In this case, we only compare the execution time of the transitive reduction by removing all Apache Spark startup and shutdown times and I/O time. Summit CPU does not provide support for Apache Spark, so we only perform this comparison on Cori Haswell.

On Cori, we used `gcc-8.3.0` and the `O3` flag to compile C/C++ codes, while on Summit we used `gcc-8.1.1`. On both Cori and Summit, we used the default MPI implementation. SORA used `jdk/1.8.0_202` and `spark/2.3.0`. In the next section, we report the average runtime over 10 runs for each experiment, except for the H. sapiens dataset at low concurrency, where we report the average over three runs.

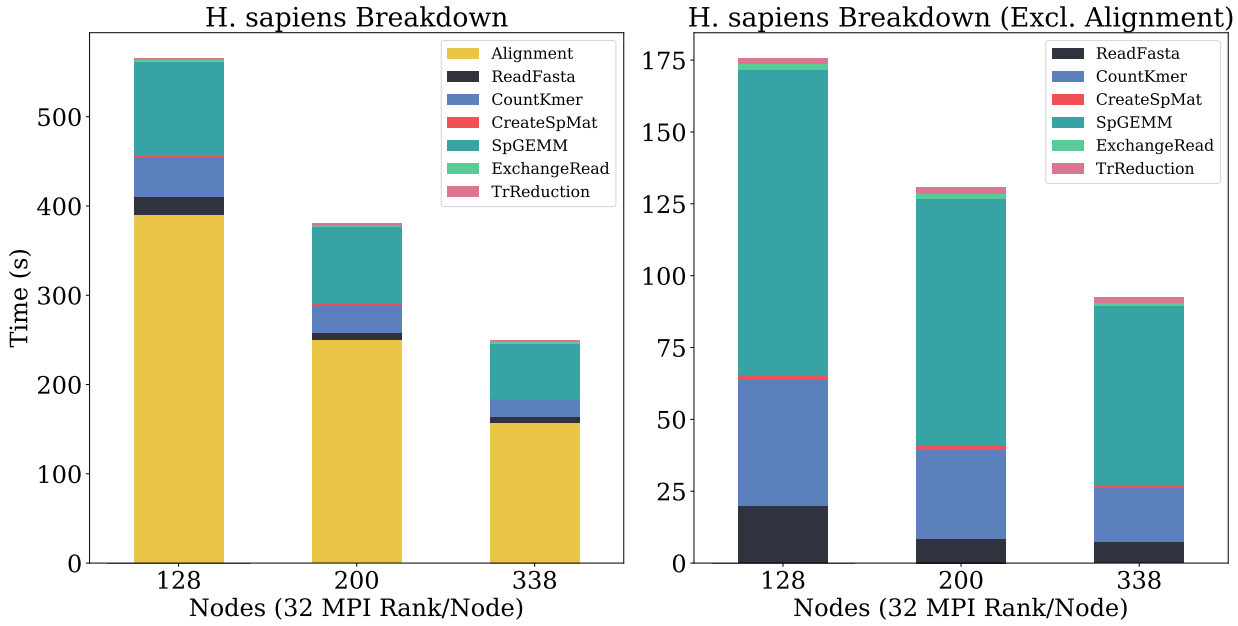

Figure 4.3: ELBA runtime breakdown on Cori Haswell (H. sapiens).

Table 4.2: Comparison of transitive reduction (in seconds) between ELBA and SORA [133] on Cori Haswell.

Dataset	Nodes	SORA	ELBA	Speed-Up
C. elegans	32	34.6	1.9	18.2×
	72	34.3	1.3	26.4×
	128	34.9	1.2	29.0×
H. sapiens	128	23.4	1.9	12.4 ×
	200	24.3	2.3	10.5 ×
	338	25.3	1.9	13.3 ×

4.5 Results

Figures 4.1 and 4.2 illustrate the performance and scaling of our transitive reduction algorithm as the top layer in each bar for C. elegans and H. sapiens on both machines. Despite a parallel efficiency of 38% on both machines, our transitive reduction shows a significant speedup compared to a competing distributed memory implementation. Figures 4.3–4.4 tell a similar story for H. sapiens.

To evaluate the competitiveness of our implementation with respect to the state of the art in distributed memoery, we compare our transitive reduction algorithm with the transitive reduction module of SORA [133], a distributed memory implementation of transitive

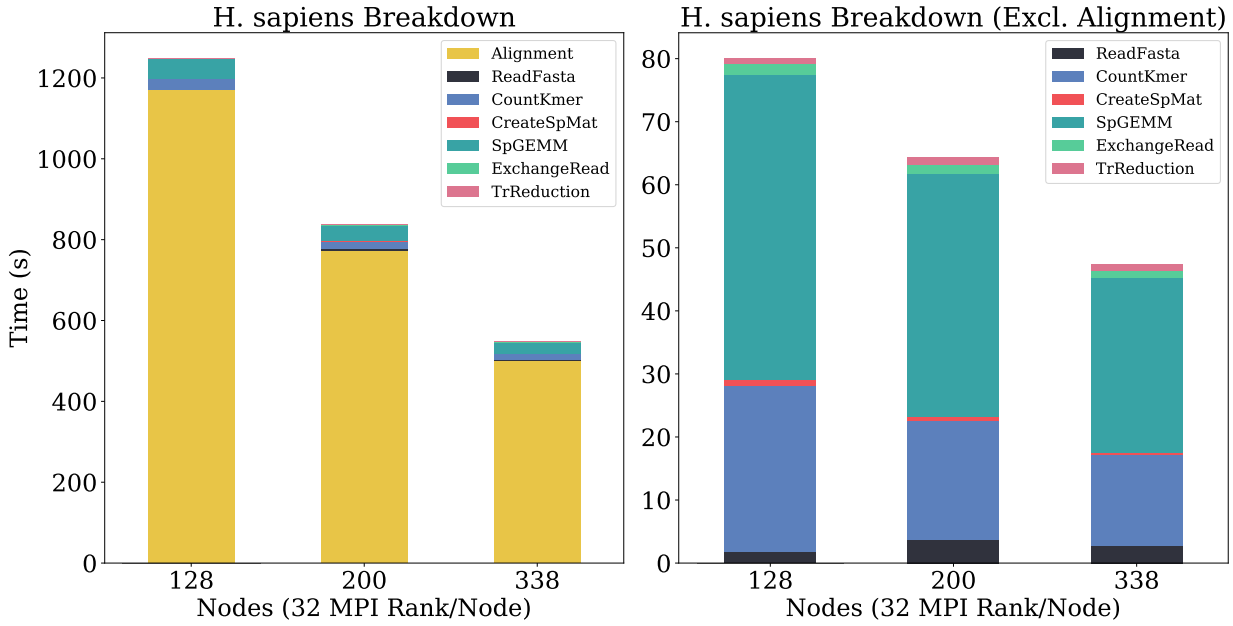

Figure 4.4: ELBA runtime breakdown on Summit CPU (H. sapiens).

reduction from overlap graph to string graph based on Apache Spark. Our transitive reduction is currently integrated into our pipeline, so in fairness we do not consider the startup, shutdown, and I/O time of SORA. The input of SORA is the overlap matrix $\mathbf{R}$ of ELBA, so both work on the same input. Our transitive reduction algorithm has a speedup of up to 29× for C. elegans and up to 13.3× for H. sapiens (Table 4.2).

4.6 Summary

Once we have identified overlap candidates in the phase of the *de novo* pipeline for genome assembly, our goal is to simplify the overlap graph into a string graph, removing any redundancy in edges and vertices. This is achieved in ELBA by introducing a novel scalable transitive reduction algorithm implemented using sparse linear algebra.

ELBA is based on linear algebraic operations over semirings using 2D distributed sparse matrices. The use of sparse matrices for both overlap detection (see Chapter 3) and transitive reduction reduces the need for various data structures normally used in genome assembly. ELBA's approach to transitive reduction resulted in a fast and efficient parallel algorithm that is up to 29× faster than a competing distributed memory implementation in Apache Spark for C. elegans and up to 13.3× faster for H. sapiens on a supercomputer.

Chapter 5

Parallel Algorithms for Contig Generation

In this chapter we describe the foundation of contig generation from the point of view of a graph, and then describe our parallel algorithms for contig generation using sparse matrices and semiring abstraction. First, we identify the branch vertices and mask them. Then, we redistribute the sequences among the processes to perform assembly locally on each process for each contig.

5.1 Overview and Foundation

In Chapter 3 we gave an overview of the Overlap step of the OLC assembly paradigm and described our approach to the computational challenge, while in Chapter 4 we described the second step of the OLC paradigm and our transitive reduction algorithm that uses sparse matrices. In this chapter, we address the final stage of the assembly pipeline, i.e., Consensus, and describe our sparse linear algebra-centered contig generation algorithm. The goal of this last phase is to extract linear sequences of vertices from the string graph and connect them to create a contig set. A contig is a continuous sequence resulting from the assembly of small DNA sequences generated by sequencing strategies, in our case long read sequencing technologies, forming a map representing a region of a chromosome.

A common approach to contig generation takes the string graph as input, branching nodes in this graph are masked, and the set of linear unbranched paths in the graph is extracted to form the contig set. Common strategies are inspired by the Bogart algorithm [100], which produces an assembly graph using a variant of the best-overlap graph strategy of Miller et al [115]. Here, the best overlap is the longest overlap to a given read end excluding contained sequences (i.e., when all bases in one sequence are aligned to another sequence). The Bogart algorithm removes overlapping sequences from the overlap graph to include only those that are within some tolerance of the global median error rate, and recalculates the longest overlapping sequences using only that subset (i.e., a sparse overlap graph). Bogart generates the initial contig set from the maximum non-branching paths in this graph. Alternatively,

Algorithm 9 Parallel matrix-based computation in ELBA.

```
1: struct ELBA
2:     sequences ← FastaReader()
3:     k-mers ← KmerCounter()
4:     A ← GenerateA(sequences, k-mers)
5:     A^T ← Transpose(A)
6:     C ← AA^T                                   ▷ Candidate overlap matrix
7:     C ← Apply(C, Alignment())                  ▷ Run alignment
8:     R ← Prune(C, AlignmentScoreLessThan(t))
9:     R ← Prune(R, IsContainedRead())
10:    S ← TransitiveReduction(R)
11:    cset ← ContigGeneration(S, sequences)      ▷ Algorithm 10
12:    return cset
13: end struct
```

Algorithm 10 Parallel contig generation on **S**.

```
1: struct ContigGeneration(S, sequences)
2:     L ← BranchRemoval(S)
3:     v ← ConnectedComponent(L)
4:     p ← GreedyPartitioning(v, P)
5:     P ← InducedSubgraph(L, p)
6:     cset ← LocalAssembly(P, sequences)
7:     return cset
8: end struct
```

some approaches generate a primary assembly based on the topological structures of the graph and the phase relationship between the different haplotypes using a bubble popping procedure [103] and generate a best overlap graph to handle the unresolved substructures.

ELBA introduces a novel distributed-memory algorithm that generates the contig set starting from a string graph representation of the genome and using a sparse matrix abstraction. ELBA first removes the branch vertices and identifies the contig set, whose associated sequences are then redistributed to allow local assemblies on each processor. Unlike MetaHipMer [69, 93], which uses fine-grained communication at nearly every step of contig construction, ELBA localizes the graph traversal problem so that the sequences that make up a contig are stored locally on each rank, avoiding communication.

This contig generation step can be followed by further polishing phases to merge and correct the contig set and to separate haplotypes.

5.2 Proposed Algorithm

ELBA uses a highly parallelizable strategy to generate the contig set in a distributed memory environment. This means that each processor may not have access to the entire set of sequences it needs for contig generation because they are distributed among processes. Read sequences must therefore be communicated across the processor grid before contigs are generated via depth-first search.

To efficiently communicate sequences where they are needed, ELBA uses a parallel algorithm, implemented as a sparse matrix-based computation over the string matrix, to extract information about which sequences belong to the same contig (i.e., a *linear component*) and therefore need to be sent to the same processor. A linear component of a graph is the maximal subgraph where each vertex is connected to two adjacent vertices, except for two vertices that mark the extremities of the linear chain, which are connected to one vertex. Once the membership of each sequence is known (i.e., to which contig it belongs), ELBA uses this information to achieve load balancing that ensures that each processor has similar amount of work during the local assembly process.

This is achieved by first estimating the contig sizes, i.e., the number of reads belonging to a particular contig, in parallel and then assigning approximately equal collections of contig sets to each process. Then, the sequences are redistributed among the processes using a newly implemented function that is also based on sparse matrix computation. This function generates local matrices from a distributed matrix (in our case, the string matrix) on each process according to the load-balanced assignment. ELBA then assembles the contigs whose sequence connectivity is stored in the local matrix in parallel on each processor without requiring any further communication.

Algorithm 9 summarizes the entire computation, including overlap detection and transitive reduction—lines 11-12 represent the main contribution of this work, better represented in Algorithm 10: (a) how the contig set is determined (lines 2-3), (b) how the workload is distributed among the processes (lines 4-5), and (c) how the local assembly is performed (line 6) and the contig set is output.

Contig Set Determination

The first step in our contig generation algorithm is to determine which sequences belong to which contig based on the connectivity of the string graph.

Let us define a linear component of the graph as the maximal subgraph $\mathbf{L} \subseteq \mathbf{R} = (V, E)$ where (a) each vertex has either degree one or two and (b) the subgraph is connected. In a linear component there are exactly two vertices which can have degree one due to the connectedness rule. These vertices are the endpoint vertices.

Then, we define the *contig set* of $\mathbf{R}$ as the set of all linear chains which are not themselves subgraphs of another linear chain in $\mathbf{R}$. Thus, the contig set consists of several independent collections of nodes in the bidirected string graph, where each collection is a linear sequence of unique overlapping sequences. The determination of the contig set is done in two stages.

First we have to identify *branching vertices* that are vertices whose degree is ≥ 3. These vertices would make it impossible to determine a unique linear chain. Therefore, our goal is to mask out such branching vertices to obtain only linear sequences of vertices. For example, consider a string graph consisting of (a) $v_1 \rightarrow v_2 \rightarrow v_3$, (b) $v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6$, and (c) $v_3 \rightarrow v_7 \rightarrow v_8$, the vertex v_3 is a branching vertex since its degree is equal to three, and it would not allow us to extract a linear chain, since at this point in the computation we cannot know whether the correct linear path is (i) $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6$ or (ii) $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_7 \rightarrow v_8$. Thus, we mask out v_3 to obtain three linear sequences of vertices: (i) $v_1 \rightarrow v_2$, (ii) $v_4 \rightarrow v_5 \rightarrow v_6$, and (iii) $v_7 \rightarrow v_8$.

To identify branching vertices, we perform a summation reduction over the row dimension of the string matrix (i.e., an adjacency matrix) and return a distributed vector $\mathbf{d}$ whose values represent the degree of the corresponding row sequence (i.e., the index of the vector). Then we perform an element-wise selection operation on the degree vector $\mathbf{d}$ to extract the indices (sequences) whose value is greater than or equal to 3. The result is a distributed vector $\mathbf{b}$ whose values are the indices of the branching vertices. The branch vector $\mathbf{b}$ is used to remove the corresponding rows and columns from the string matrix $\mathbf{S}$ and create a linear chain version of $\mathbf{S}$ that we name $\mathbf{L}$ (line 2, Algorithm 10). From a graph-theoretic point of view, this operation removes (i.e., sets to zero) the edges adjacent to branching vertices. The indexing of the matrix does not change, but its nonzeros do. For example, if row 10 is a branching vertex, the entire row—and column, since $\mathbf{S}$ is symmetric—is cleared, but row 10 is still a row in the matrix. This way we can avoid re-indexing sequences during the computation. The result is an even sparser string graph with nodes of degree 0, 1 or 2. For each contig, there are exactly two vertices of degree 1, which we call root nodes (or root vertices) and use as starting points for depth-first search and local assembly of the contig in the final stage of contig generation.

Once the string matrix is in its unbranched form $\mathbf{L}$, we want to decompose it into its linear components to produce manageable independent subproblems that we can work on in parallel (i.e., local assemblies). Therefore, we use the sparse matrix based connected components (LACC) algorithm presented by Azad et al. [8] to determine the contig set. LACC is a distributed-memory implementation of the Awerbuch-Shiloach algorithm [7] using the CombBLAS library. LACC takes advantage of the sparsity of the vector to avoid processing inactive vertices. It takes as input the unbranched sparse matrix $\mathbf{L}$ and returns a distributed vector $\mathbf{v}$ (line 3, Algorithm 10), which is a mapping from global sequence indices (i.e., rows and columns of $\mathbf{S}$) to contig indices C_i. In the previous graph example, LACC would return $v_1, v_2 \in C_1$, $v_4, v_5, v_6 \in C_2$, and $v_7, v_8 \in C_3$.

Once we have determined which sequences belong to which contig, we estimate the size of each contig by counting how many sequences belong to it. The exact size depends on the lengths of the sequences (minus their overlapping areas), but for simplicity we estimate it using only the number of sequences (i.e., vertices), since virtually every read and overlap could have a different length. However, since $\mathbf{v}$ is a distributed vector, a processor may not know the full set of sequences belonging to the contig it owns. Therefore, we need to communicate this information across the processor grid. Each processor computes a local

size estimate based on the vertices it owns locally for each local contig. An MPI Reduce-scatter collective operation is used to determine the global size of each contig and redistribute the sizes across the processor grid to create a distributed mapping of contig indices to their associated global sizes.

Contig Load Balancing and Communication

Previously, we determined which sequences belong to which contig and estimated the size of each contig based on the number of sequences associated with it. Using the contig size as an optimization parameter, we want to distribute the workload as evenly as possible across the processor grid.

Load Balancing Algorithm Given a vector of contig sizes of length n and P processes, we want to find the near-optimal contig-to-processor assignment such that the amount of work estimated by the contig size that each processor has to do is similar. Our objective and problem definition is similar to the *multiway number partitioning* optimization problem first introduced by Ronald Graham in the context of the identical-machines scheduling problem [79]. The classical application is to schedule a set of m jobs with different runtimes on k identical machines in such a way that the makespan, i.e., the elapsed time until the schedule is completed, is minimized.

In the context of contig generation, this means that given a multiset of S instances (in our case, the contig sizes), we want to partition this multiset into P subsets (the number of processes in our processor grid and must be a positive integer) such that the sums of the subsets (i.e., the sums of the contig sizes) are as similar as possible. The partitioning results in a balanced distribution of the workload for the next and final phase of the computation.

The multiway number partitioning problem is NP-hard. To overcome this limitation, in ELBA we use an approximation algorithm known in the scheduling literature as the Longest Processing Time (LPT) algorithm, whose goal is to minimize the largest subset, which belongs to a class of algorithms known as *greedy number partitioning*. The algorithm loops over the contig sizes and inserts each number into the set whose current sum of sizes is smallest. The result is a partitioning that minimizes the time processes spend waiting for the most heavily loaded process to finish assembling its contig subset. If the contig sizes were not sorted, then the runtime would be $O(n)$ and the approximation ratio would be at most $2 - 1/P$. It is possible to improve the approximation ratio to $(4P - 1)/3P$ by sorting the input vector of contig sizes [80].

The improvement in the approximation using LPT is accompanied by an increase in the runtime to $O(n\log n)$ due to sorting. However, the number of contigs n is smaller than the number of sequences by at least an order of magnitude, and the increase in runtime does not create a computational bottleneck. For the same reason, we collect the global information about contig lengths in a single processor and run the partitioning algorithm on it to avoid the unnecessary communication of small messages. The partitioning algorithm returns the vector $\mathbf{p}$ specifying the assignment of contigs to processes (line 4, Algorithm 10)—$\mathbf{p}$ is broadcast

to the entire processor grid so that each local process can determine where to send its local sequences and associated information.

As mentioned earlier, the problem size at this stage of the computation is often smaller than the problem size at the initial stage (i.e., overlap detection), so for some species it is possible that $n < P$. In this case, some of the processes are idle for the final phase of the computation. For the two species we use in the experimental evaluation in Section 5.3, n is equal to 6411 and 4287 and P varies from 18 to 128. In the next section, we explain how contigs are redistributed among processes based on their size.

Induced Subgraph Algorithm Once we have determined where a contig and its sequences should be stored, we must perform the actual communication step to send the linear component information and associated sequences to the owner processor.

Communicating both the linear component information and the sequences involves the same high-level procedure of reassigning vertices representing a linear chain to their owner processor. However, the underlying data structures are fundamentally different, namely that the overlap graph is stored as a sparse matrix while the sequences are stored as distributed char arrays, and therefore require a different implementation.

Let us first focus on the communication of linear component information, i.e. the graph-like structure that stores connectivity information. We have as input the sequence-by-sequence matrix $\mathbf{L}$, the resulting matrix after the transitive reduction step and from which we have cut out vertices of degree ≥ 3. $\mathbf{L}$ is distributed over P processes, which are logically organized in a $\sqrt{P} \times \sqrt{P}$ grid. Let n be the number of vertices in $\mathbf{L}$, where $\mathbf{L} = (V, E)$ in the graph interpretation. From the load balancing algorithm, we have a distributed vector $\mathbf{v} : [n] \rightarrow [P]$ such that $\mathbf{v}[u] = P_i$ means that vertex u should belong to processor P_i.

The goal is to create an induced subgraph—or submatrix—$\mathbf{L}^{(P_i)}$ locally on each processor P_i, i.e., a graph formed by a subset of the vertices of the original graph $\mathbf{L}$ and the edges connecting the vertices in this subset. Formally, we define an induced subgraph $\mathbf{L}^{(P_i)}$ such that $\mathbf{L}^{(P_i)} = (V^{(P_i)}, E^{(P_i)})$ with $V^{(P_i)} = \{v \in V : P_i = \mathbf{v}[v]\}$ and $E^{(P_i)} = \{(u, v) \in E : u, v \in V^{(P_i)}\}$.

The vector $\mathbf{v}$ is also distributed across the $\sqrt{P} \times \sqrt{P}$ processor grid and is therefore divided into P subvectors, each of size $\approx n/P$; note that we write $\mathbf{v}_{(i,j)}$ to denote the subvector on process $P(i, j)$. Each process is only aware of which vertices it stores to send to other processes. Therefore, we need a communication step to make each process aware of which vertices to receive. Given the way $\mathbf{v}$ is distributed, we can avoid an MPI_Allgather operation spanning the entire grid and instead use the square process grid to communicate $\mathbf{v}$ in a scalable way. That is, we perform an allgather operation over the **Row** dimension followed by point-to-point communication to access the information stored on the **Column** dimension.

More precisely, let $(u, v) \in \mathbf{L}_{i,j}$ be any nonzero value, where $\mathbf{L}_{i,j}$ is the local submatrix stored on processor $P(i, j)$. The goal is to determine $\mathbf{v}[u]$ and $\mathbf{v}[v]$. Given we know (u, v) is stored in the submatrix $\mathbf{L}_{i,j}$, we know that $\mathbf{v}[u]$ is stored in the row $P(i, :)$. Thus, we

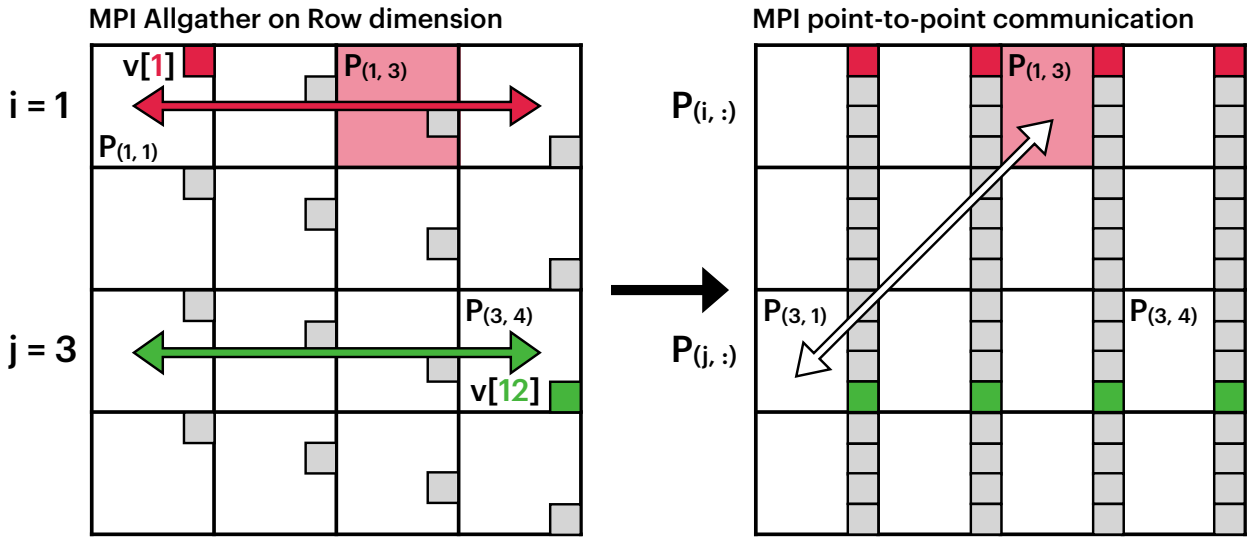

Figure 5.1: An example of how the induced subgraph algorithm communicates vertices across the processor grid.

can access $\mathbf{v}[u]$ by computing an MPI_Allgather operation over the processor row $P(i,:)$ and, more generally, over the **Row** dimension of $\mathbf{S}$. To determine $\mathbf{v}[v]$, we first note that if we are a diagonal processor, i.e. $i = j$, we already have access to $\mathbf{v}[v]$ via the previous MPI_Allgather operation. If $i \neq j$, then we know that $\mathbf{v}[v]$ is stored on the transposed processor $P(j,i)$ because we performed the allgather operation over the **Row** dimension, i.e. $P(j,:)$. Therefore, we perform a point-to-point communication between $P(i,j)$ and $P(j,i)$ to exchange the subvector $\mathbf{v}_{i,*}$ and the subvector $\mathbf{v}_{j,*}$. This gives the processor $P(i,j)$ access to (u,v). It follows that each processor $P(i,j)$ now has access to every entry in $\mathbf{v}$ corresponding to a nonzero in $\mathbf{L}_{i,j}$. A similar procedure is previously implemented for distributed-memory breadth-first search [30].

Figure 5.1 shows an example with 1-based indexing, where $u = 1$ (red square in the leftmost matrix) is stored on process $P(1,1)$ and $v = 12$ (green square) is stored on process $P(3,4)$. If we assume that $P(1,3)$ requires access to $\mathbf{v}[1]$ and $\mathbf{v}[12]$, an MPI_Allgather operation on the $P(1,:)$ row provides access to $\mathbf{v}[1]$, while the same operation on the $P(3,:)$ row provides access to $\mathbf{v}[12]$ on the transposed processor $P(3,1)$. Point-to-point communication between $P(1,3)$ and $P(3,1)$ then enables $P(1,3)$ to access $\mathbf{v}[12]$.

Once we have access to the processor assignment stored in $\mathbf{v}$, we communicate edges to their target processor and build the local induced subgraph. To do this, we loop through each nonzero $(u,v) \in \mathbf{L}_{i,j}$ stored on processor $P(i,j)$, and for each (u,v) where $\mathbf{v}[u]$ and $\mathbf{v}[v]$ are destined for the same destination processor, we construct a triple $(u,v,\mathbf{S}(u,v))$ and place it on an outgoing buffer to the destination processor. A custom all-to-all ensures the required re-distribution of non-zeros. Once each processor has access to its edge set $E^{(P_i)}$,

the local adjacency matrix of the induced subgraph is constructed; while we re-index the local matrix to fit its new, smaller size, we also keep a map of the original global vertex indices, since it is needed in the final phase.

Read Sequence Communication The communication of read sequences is implemented separately, since the read sequences are not stored as nonzeros in the sparse matrix $\mathbf{L}$, but in a distributed auxiliary data structure. Furthermore, a sequence is represented and stored as a char array. A large dataset could exceed the MPI count index limit of $2^{31} - 1$ because the implementation is a 4-byte integer. Therefore, we need to treat sequence communication separately and consider this potential limitation.

Read sequences that need to be sent to a processor other than the one they are currently on are packed into a char buffer and communicated as a sequence of non-blocking point-to-point messages in an all-to-all fashion. To deal with the MPI $2^{31} - 1$ count limit, we check the length of each message to be communicated and its receive buffer. If it goes beyond the limit, we communicate the sequences using a user-defined contiguous MPI data type whose size is equal to the buffer length. This way we can send and receive each character buffer in a single call.

Local Contig Assembly

Let $\mathbf{L}_{i,j}$ be the local graph (or matrix), composed of one or more linear components, stored on the processor $P_{i,j}$ obtained via the induced subgraph algorithm. There are P such graphs, one for each processor, but we can assume without loss of generality that we are dealing only with $\mathbf{L}(P_{i,j})$ or $\mathbf{L}$ for short.

Suppose $\mathbf{L}$ has n vertices and m edges. Each vertex of $\mathbf{L}$ is a read sequence l assigned by the partitioning algorithm to $P_{i,j}$. Then, we define $\Sigma = \{A, C, T, G\}$ as the alphabet of DNA nucleotides. Consequently, we represent the n sequences of $\mathbf{L}$ by $l_0, l_1, \ldots, l_{n-1} \in \Sigma^*$, which means that each sequence is a combination of $\{A, C, T, G\}$. Given any read sequence l, we express the nucleotides of that sequence by $(l[0], l[1], \ldots, l[|l| - 1])$, where $|l|$ denotes the length of the sequence. If $l[i]$ is a base or nucleotide in Σ, we denote the Watson-Crick complement base of $l[i]$ by $l[i]^c$. This allows us to generalize the notion of sequence to include its reverse complement, as defined in Chapter **??**: if $i < j$, then we write $l[i : j]$ for the substring $(l[i], l[i + 1], \ldots l[j])$ and $l[j : i]$ for its reverse complement substring $(l[j]^c, [j - 1]^c, \ldots, l[i]^c)$.

$\mathbf{L}$ can also be viewed from the point of view of its matrix representation, such that $\mathbf{L}$ is a sparse $(n \times n)$ matrix with m nonzeros. In the earlier stages of the pipeline, we use the doubly compressed sparse column (DCSC) format [25] to store our matrices for scalability. However, for the local traversal algorithm described in this section, we converted these matrices to compressed sparse column (CSC) format for simplicity and faster vertex (column) indexing. The storage space required to store the local matrices is an order of magnitude smaller than before and hence CSC does not introduce memory scalability issues despite hypersparsity. This conversion takes linear time in the number of local vertices, as

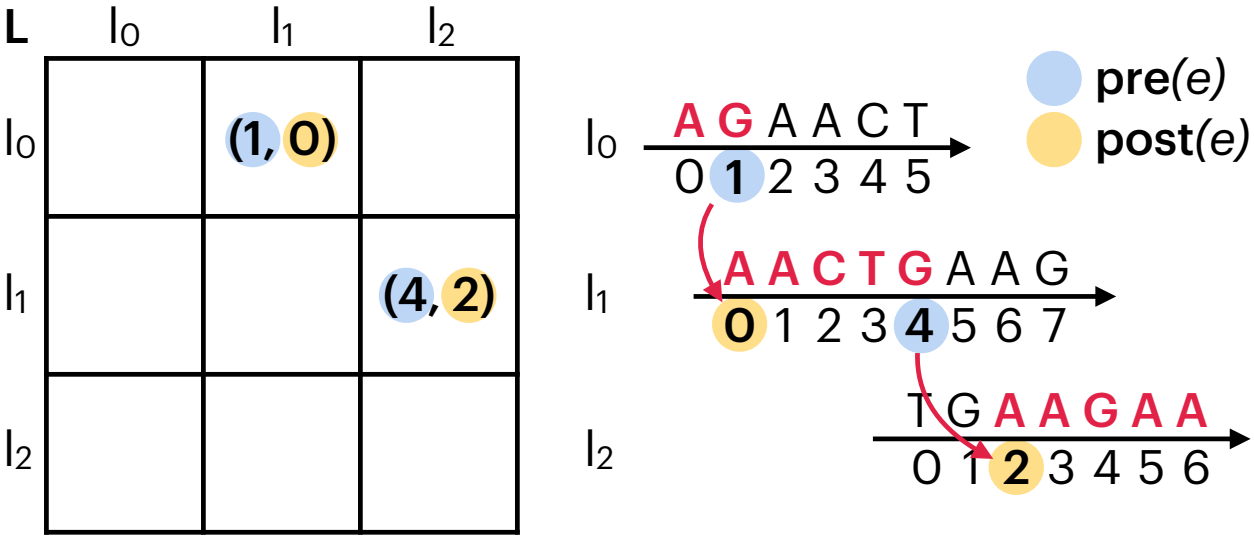

Figure 5.2: An example of how the local assembly algorithm concatenates sequences.

only column pointers needs to be uncompressed and row indices array stays intact, and has negligible effect on overall runtime.

As a sparse matrix, $\mathbf{L}(u, v) = e$ is a nonzero provided that the source sequence l_u and the destination sequence l_v overlap. In this nonzero e we store two values computed from the original string graph $\mathbf{S}$, which is all we need for assembly: $\mathbf{pre}_u(e)$ and $\mathbf{post}_v(e)$. To define $\mathbf{pre}_u(e)$ and $\mathbf{post}_v(e)$, we use inclusive indexing, resulting in the following asymmetric definition. The first value $\mathbf{pre}_u(e)$ stores the index i of l_u, which is the last nucleotide in l_u that does not overlap with l_v, i.e., the nucleotide on l_u that precedes the overlap with l_v. The second value $\mathbf{post}_v(e)$ stores the index j of l_v, which is the first nucleotide in l_v that overlaps with l_u, i.e., the index of the beginning of the overlap between l_u and l_v. The definition is asymmetric because $l_u[i]$ and $l_v[j]$ do not store the same nucleotide. This asymmetry is necessary to compute the non-overlapping prefixes of each read in a contig, which, when joined together, produce a contig sequence.

For example, if $l_u = l_0 = AGAACT$ and $l_v = l_1 = AACTGAAG$ as shown in Figure 5.2 and we consider 0-based indexing, then $\mathbf{pre}(e) = 1$ and $\mathbf{post}(e) = 0$ because $l_u[2:5]$ and $l_v[0:3]$ overlap. If instead we consider $l_u = l_1 = AACTGAAG$ and $l_v = l_2 = TGAAGAA$, then $\mathbf{pre}(e) = 4$ and $\mathbf{post}(e) = 2$ because $l_u[5:7]$ and $l_v[2:4]$ overlap. It is also possible that $l_u = l_0$ would overlap with $l_v^c = l_1^c = CTTCAGTT$ (the reverse complement of l_1). In this case, the same procedure would apply to compute $\mathbf{pre}(e) = 1$ and $\mathbf{post}(e) = 4$, where the overlapping subsequences would be $l_u[2:5]$ and $l_v^c[7:4]$, since the l_u subsequence $AACT$ would overlap with the l_v^c reserve complement subsequence $AGTT$.

Theoretically, the beginning of the overlapping substring of l_v should always be 0 or $|l_v| - 1$, depending on the orientation. Yet, we still store $\mathbf{post}(e)$ because the diBELLA

2D [85] pipeline, which we extend in this work, uses an x-drop seed-and-extend approach to align overlapping sequences. With x-drop, the alignment between two sequences can potentially end early (i.e., before extending to the end of a read), leaving a short overhang in the alignment coordinates at the end of the sequence. If $l_u = l_1 = AACTGAAG$ and $l_v = l_2 = TGAAGAA$ are defined as in Figure 5.2, and the alignment told us that $l_u[5:7]$ and $l_v[2:4]$ were the overlapping region, it would be incorrect to match $\mathbf{pre}(e) = 4$ with $\mathbf{post}(e) = 0$, since we need $\mathbf{post}(e) = 2$ to correctly concatenate the subsequences that form a contig. If we join the three partial sequences marked in red on l_0, l_1, l_2 in Figure 5.2, as explained below, we get a contig.

Because the local assembly operates directly on the CSC format, we describe it briefly. $\mathbf{L}$.JC is the column pointer array of length $n+1$, $\mathbf{L}.IR$ is the row index array of length m, and $\mathbf{L}.VAL$ is the array of tuples $(\mathbf{pre}(e), \mathbf{post}(e))$, also of length m.

The local assembly algorithm is a variant of depth-first search, simplified by the fact that the maximum vertex degree of $\mathbf{L}$ is 2 by construction. For this reason, there is always only one vertex in the frontier, and the search is thus a linear walk. Each local matrix $\mathbf{L}_{i,j}$ is assigned a contig set by multiway number partitioning, which is the set of connected components of $\mathbf{L}_{i,j}$. Each vertex of $\mathbf{L}$ has degree 1 or 2. Moreover, we define contigs as linear chains of at least two sequences, where q is the number of vertices in a given connected component. It follows that any connected component consisting of $q \geq 2$ vertices must have exactly 2 vertices of degree 1 (i.e., *root* vertices) and $q-2$ vertices of degree 2 (i.e., *intermediate* vertices). In short, the idea is to scan for root vertices in $\mathbf{L}$ and, if we find them, to take a walk from that root vertex (via as many intermediate vertices as necessary) until another root vertex is found. As we proceed, we perform the concatenation of subsequences described below. This search for the root vertex is performed over all n vertices. Therefore, we must mark each final root vertex (found by the linear traversal) as visited to avoid accidentally composing the same contig twice.

More precisely, we loop over the n sequence vertices in each subgraph $\mathbf{L}_{i,j}$ and compute $\mathbf{L}_{i,j}.JC[i+1] - \mathbf{L}_{i,j}.JC[i]$ from the column pointer array representing the degree of vertex i. Each time we find a vertex of degree 1, i.e., a root vertex, that has not been visited before, we perform a traversal starting from this that root vertex r. Given a generic vertex in the chain c (which can be either a root vertex or an intermediate vertex), we find the successor vertex by examining the vertices in the slice $\mathbf{L}_{i,j}.IR[\mathbf{L}_{i,j}.JC[c] : \mathbf{L}_{i,j}.JC[c+1]]$ of the row index array where the edges are stored. Each vertex c has at most two successor vertices, and we select the unvisited one. This process continues until we reach the vertex r' for which $\mathbf{L}_{i,j}.JC[r'+1] - \mathbf{L}_{i,j}.JC[r'] = 1$, i.e., the second root vertex of a contig. The result is a chain of q vertices $r, c_1, \ldots, c_{q-2}, r'$ for each contig stored in the submatrix $\mathbf{L}_{i,j}$. As we proceed, we collect the edges between them $e_0, e_1, \ldots, e_{q-2}$. Because the values for $\mathbf{pre}(e_i)$ and $\mathbf{post}(e_i)$ have already been computed and stored in each edge e_i, we know exactly which subsequences of which read sequence to look up and join to form the contig. Namely, $l_r[\alpha : \mathbf{pre}(e_0)] \oplus l_{c_1}[\mathbf{post}(e_0) : \mathbf{pre}(e_1)] \oplus \ldots \oplus l_{c_{q-2}}[\mathbf{post}(e_{q-3}) : \mathbf{pre}(e_{q-2})] \oplus l_{c_{r'}}[\mathbf{post}(e_{q-2}) : \beta]$, where $\alpha = 0$ or $|l_r| - 1$ depending on the orientation of l_r, and β is similarly defined for $l_{r'}$.

Table 5.1: Data sets used during evaluation: name, depth, number of sequences in the input, average read length, input size, genome size, and error rate.

Label	Depth	Reads (K)	Length	Input (GB)	Size (Mb)	Error Rate (%)
O. sativa	30	638.2	19,695	12.2	500	0.5
C. elegans	40	420.7	14,550	3.8	100	0.5
H. sapiens	10	4,421.6	7,401	31.1	3,200	15.0

This algorithm is performed by each process in parallel on its own induced subgraph $\mathbf{L}_{i,j}$. For each read in the contig, we either look for the subsequence in the locally stored char array that we started with, or in the char array obtained by communicating the read sequence. Because we store read sequences in large packed char arrays, we do not need to copy the entire read sequence to find the correct subsequence. Instead, we can simply use the offsets already computed, which tell us where each read is in the buffer, and then read the subsequence directly from the buffer. The algorithm is $O(q)$, where q is the number of vertices in a connected component as previously defined (i.e., the number of sequences in a contig), since the search for the root vertex takes linear time and the traversal is linear in the number of edges, which is $2(q-1)$.

5.3 Experimental Setup

To evaluate our contig generation algorithms and the ELBA long-read assembly pipeline we used the same two machines used previously: the Haswell partition of the Cray XC40 supercomputer Cori at NERSC and the IBM supercomputer Summit at Oak Ridge National Laboratory, on which we used only IBM POWER9 CPUs (Table 3.5).

To evaluate our algorithm, we use three different species: Oryza sativa (O. sativa), Caenorhabditis elegans (C. elegans), and Homo sapiens (H. sapiens) whose characteristics are summarized in Table 5.1. The evaluation is divided into two categories: (a) runtime and scalability of the ELBA pipeline including our novel contig generation algorithm on two machines, and (b) runtime compared to state-of-the-art shared memory software. For low error rate sequences (O. sativa and C. elegans), we also report assembly quality.

The state-of-the-art software we consider are Hifiasm [38] and HiCanu [127], because of their speed and popularity, respectively, and both are written for shared-memory parallelism. HiCanu can optionally run on grid computing, but is not implemented for distributed memory parallelism. It is worth noting that Hifiasm and HiCanu include additional polishing stages that make their overall assembly quality higher than ELBA's. For the H. sapiens dataset, we consider Miniasm [103] and Canu [100], as this dataset has a much higher error rate that is not suitable for Hifiasm and HiCanu. Our goal is to show the competitiveness and potential of ELBA in terms of assembly quality, demonstrating in particular clear advantages in terms of runtime. ELBA was run with the k-mer length parameter $k = 31$ and the x-drop threshold

x = 15 for the low error rate data and with k = 17 and x = 7 for H. sapiens in Table 5.1. Hifiasm and HiCanu were run with their default setting.

To show performance, we run ELBA on both Cori Haswell and Summit (except for H. sapiens, where we only use Summit), while for comparison with the state of the art, we only use Cori Haswell, since Hifiasm and HiCanu use SSE and AVX2 intrinsics, which are not supported on the IBM POWER9 processor on Summit. Hifiasm and HiCanu were developed for shared memory, so we only give runtimes for a single Cori Haswell node using multi-threading. The lack of support for AVX2 intrinsics is also the reason why ELBA's alignment is slower on Summit than on Cori. For this reason, we want to emphasize that the goal of using two machines is to show performance and scaling on different systems, not to directly compare the two machines, since ELBA is optimized for a general HPC system, i.e., our code is general and no architecture-specific optimizations have been made.

To assess the contig quality of of ELBA, Hifiasm, and HiCanu, we use QUAST [87] and report the following metrics: completeness, longest contig size, number of contigs, and misassembled contigs. Completeness measures the percentage of the reference genome to which at least one contig has been aligned. This is calculated by counting the number of nucleotides aligned to the reference genome and dividing by the total length. The number of misassembled contigs is defined as the number of contigs that contain incorrect assemblies, e.g., a contig consisting of sequences originating from different regions of the reference genome.

5.4 Results

In this section, we evaluate the performance and quality of the overall ELBA pipeline and our novel contig generation algorithm, both individually and compared to the literature.

Figure 5.3 illustrates the strong scaling of the entire ELBA pipeline for C. elegans on the left and for O. sativa on the right. The C. elegans dataset was run on P = {18, 32, 50, 72, 128} nodes using 32 MPI ranks/nodes on both machines, while the O. sativa was run on P = {18, 32, 50, 72, 128} nodes using 32 MPI ranks/nodes on Summit CPU and on P = {50, 72, 128} nodes Cori Haswell because the algorithm for P = {18, 32} ran out of memory since Cori Haswell has a smaller memory per node than Summit.

ELBA achieves a parallel efficiency of 75% on Cori Haswell and 69% Summit CPU for C. elegans, while it achieves a parallel efficiency of 80% and 64% for O. sativa. The parallel efficiency of O. sativa between 72 and 128 nodes on Summit is 83%, which is similar to the parallel efficiency on Cori between 50 and 128 nodes. For the H. sapiens dataset, the parallel efficiency on Summit between 200 and 392 nodes is close to 90%, as shown in Figure 5.5 on the left. These results show a good scaling behavior of the whole ELBA pipeline with a large input on two different architectures.

Figure 5.4 shows the runtime breakdown of ELBA on the two machines for C. elegans on the left and for O. sativa on the right, while Figure 5.5 on the right shows the runtime breakdown of ELBA on Summit for H. sapiens. To highlight the impact of the major stages,

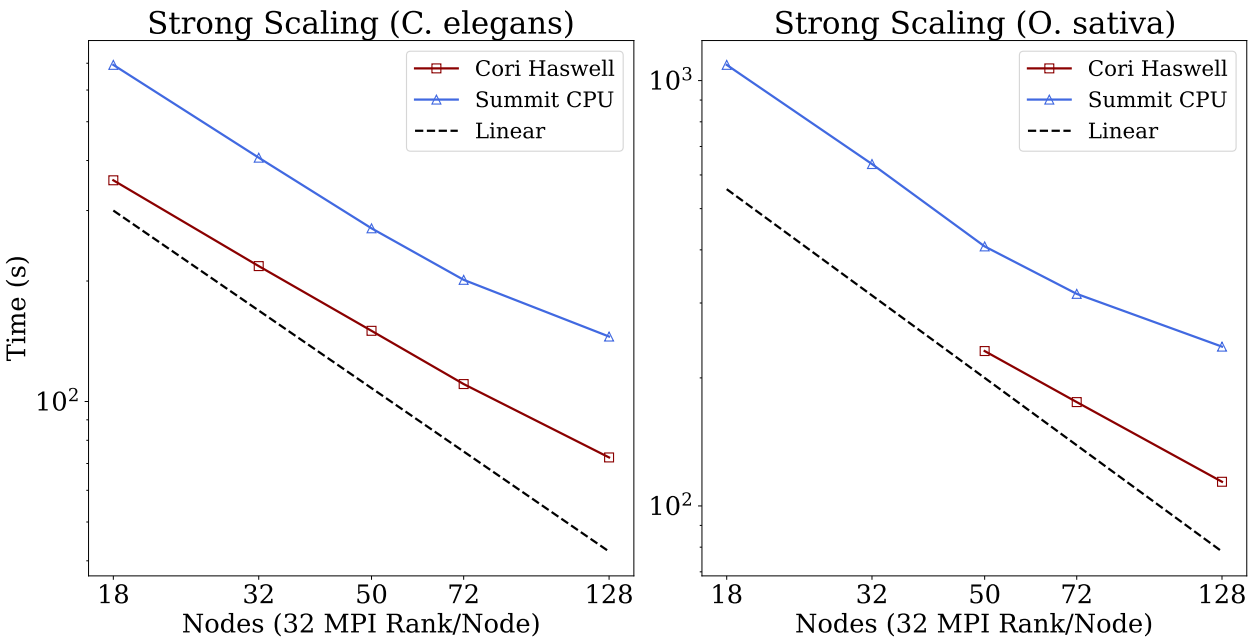

Figure 5.3: ELBA strong scaling on Cori Haswell and Summit CPU using 32 MPI rank/node on C. elegans (left) and on O. sativa (right).

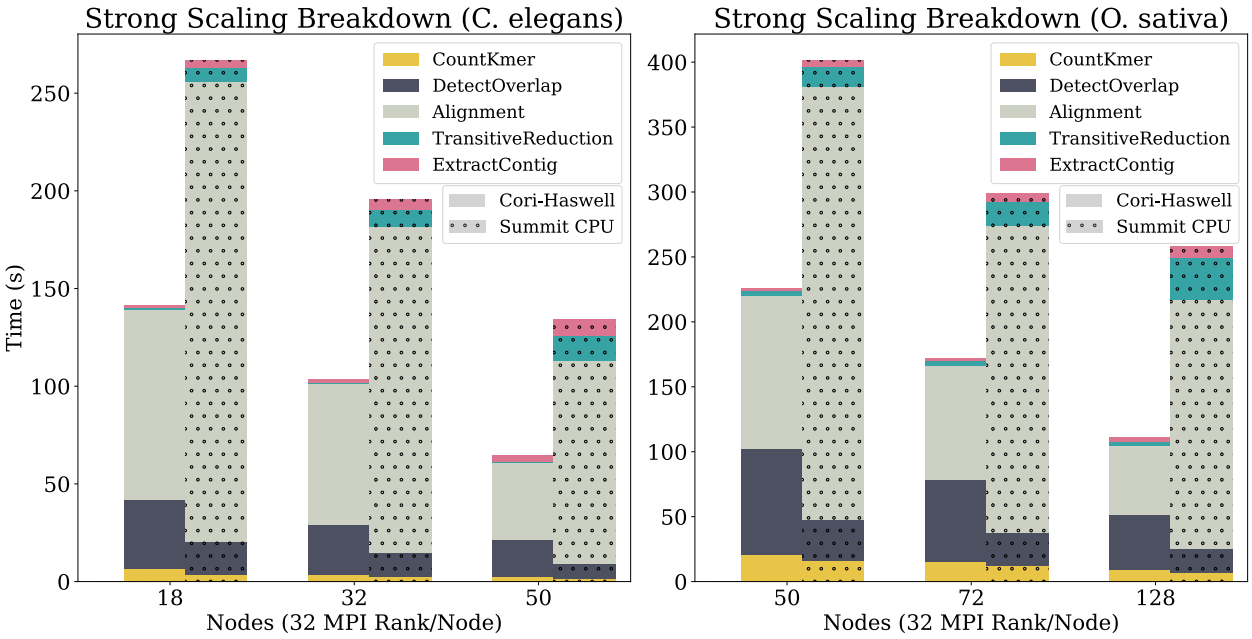

Figure 5.4: ELBA runtime breakdown of the main stages of the pipeline on Cori Haswell and Summit for C. elegans on the left and for O. sativa on the right.

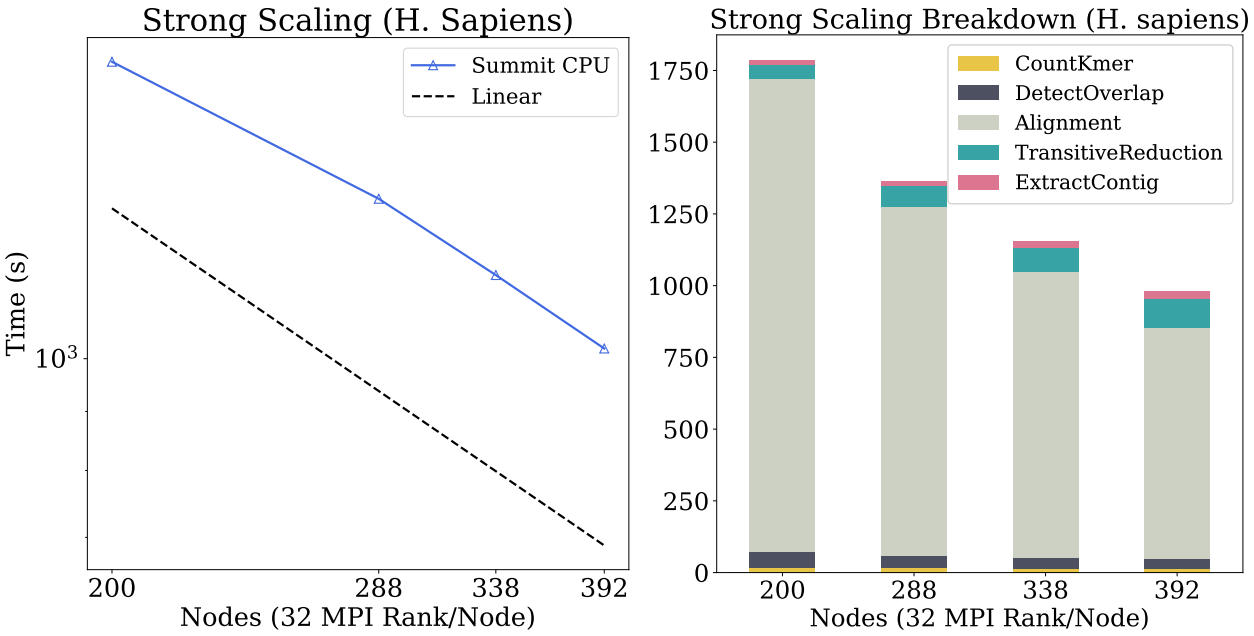

Figure 5.5: ELBA strong scaling (on the left) and runtime breakdown (on the right) of the main stages of the pipeline on Summit for H. sapiens.

we omit I/O and other minor computation from the breakdown. The overall impact of the omitted computation is negligible.

The legend is arranged in reverse order with respect to the layers of the bar, i.e. the first entry of the legend at the top is associated with the first layer in the stacked bar from the bottom. `CountKmer` corresponds to the k-mer counting step, then follows the step `DetectOverlap`, which represents the time to create and compute the candidate overlap matrix $\mathbf{C}$. Then comes the `Alignment` step, i.e., the time required to perform pairwise alignment on each nonzero in the candidate overlap matrix $\mathbf{C}$, followed by `TrReduction`, i.e., the transitive reduction time. Finally, `ExtractContig` is the time we spend extracting the contig set from the sparse matrix $\mathbf{S}$. The `ExtractContig` step is the core contribution of this work, but its implementation is an essential part of making the entire ELBA pipeline work and for this reason it is key to showing the scalability of the pipeline.

Figure 5.4 shows the breakdown of ELBA performance for the main phases of computation for C. elegans and O. sativa using P={50, 72, 128} nodes on each machine while Figure 5.5 shows the breakdown of ELBA performance breakdown for H. sapiens using P={200, 288, 338, 392} Summit nodes. ELBA is faster overall on Cori Haswell than on Summit CPU. The `CountKmer`, `DetectOverlap`, and `Alignment` phases show nearly linear scaling on both machines, with the exception of `CountKmer` for H. sapiens on Summit, which scales only sublinearly. The relative contribution of pairwise alignment to the overall runtime increases on Summit CPU compared to Cori Haswell and largely contributes to the discrep-

ancy in runtime because the alignment library used in ELBA is not optimized for the IBM processor. The `TrReduction` and `ExtractContig` phases are also significantly faster on Cori Haswell than on Summit CPU. On Summit, the computation for these two phases does not scale and consumes a higher percentage of runtime. This is because the amount of work is smaller in these two phases and the algorithms are latency-bound. Further, Summit's network has lower performance compared to Cori Haswell's. Summit CPU has lower network bandwidth per core, and we also only use 32 of the 42 available cores on Summit to make the comparison with Cori fair, which does not saturate Summit's bandwidth. In the future, we plan to make the pipeline modular so that a user can run different phases separately (incurring some I/O overhead). This would allow the less compute and memory intensive phases to be run on a smaller number of nodes without sacrificing performance.

In each dataset, 65-85% of the runtime of contig generation on both machines is taken by the induced subgraph function described in Section 5.1, which mainly involves communication. The `ExtractContig` never requires more than 5% of the computation for each species and each machine, demonstrating the efficiency of our contig generation algorithm.

ELBA's contig generation focuses on localizing the graph traversal problem so that the sequences that make up a contig are organized into localized matrices on each processor. The working set at this stage is smaller than at the beginning of the computation, so the result is a fast and efficient, but latency-bound algorithm.

To demonstrate the competitiveness and advantages of ELBA over state-of-the-art long read assembly software, we compare the runtime and scaling of ELBA with Hifiasm [38] and HiCanu [127] for O. sativa and C. elegans and with Mifiasm [103] and Canu [100] for H. sapiens. The competing software are designed for shared memory parallelism. Therefore, we ran them on a single Cori Haswell node with multithreading, while we ran ELBA on $P = \{18, 50, 128\}$ with 32 MPI ranks/nodes. Due to the hard-to-separate differences among this software, we decided to make a comparison based on total runtime. Hifiasm, HiCanu, Mifiasm, Canu, and ELBA perform very different computations, but they ultimately aim to solve the same problem. For the sake of completeness, we also compare the quality of assembly. It is worth noting that both Hifiasm and HiCanu perform a polishing phase. This can lead to a slight disadvantage in runtime, but ensures a better assembly quality.

One could consider coupling ELBA with a state of the art polishing software. However, this type of computation is highly dependent on the previous stages, so coupling our pipeline with software not specifically designed for this purpose would not allow a fair comparison. Therefore, the development of an ad-hoc polishing software for ELBA remains a future work.

Table 5.2 summarizes the runtime performance of Hifiasm and HiCanu, in the rightmost column the speedup of ELBA over those software for $P = \{18, 128\}$ and $P = \{50, 128\}$ for C. elegans and O. sativa, respectively. ELBA is up to 15× faster than Hifiasm and up to 58× faster than HiCanu for the C. elegans dataset, while the speedup for O. sativa, which is a larger genome, reaches 36× over Hifiasm and up to 159× over HiCanu. On Cori, runtimes for Hifiasm and HiCanu for O. sativa are 17 and 64 minutes, respectively, while ELBA takes less than 2 minutes on 128 nodes. For C. elegans, runtimes of Hifiasm and HiCanu are approximately 1 hour and 5 hours, while ELBA requires 1 minute on 128 nodes. For H.

Table 5.2: ELBA's speedup over state-of-the-art software. Hifiasm and HiCanu are designed for shared memory parallelism and were run on a single Cori node with multithreading.

Tool	Organism	Runtime (s)	Nodes (32 MPI Rank/Node)	ELBA Speed-Up
Hifiasm	C. elegans	1,015.2	18–128	3–15×
HiCanu	C. elegans	3,819.0	18–128	11–58×
Hifiasm	O. sativa	4,131.9	50–128	18–36×
HiCanu	O. sativa	18,131.0	50–128	78–159×

Table 5.3: Comparison of assembler quality for O. sativa (top) and C. elegans (bottom). Hifiasm and HiCanu implement additional polishing stages to improve their metrics.

Tool	Completeness (%)	Longest Contig (Mb)	Contigs	Misassembled Contigs
ELBA	37.09	0.172	6411	2
Hifiasm	26.94	7.083	1661	1
HiCanu	25.94	37.523	168	2
ELBA	98.93	0.313	4287	5
Hifiasm	99.96	6.438	133	0
HiCanu	99.90	18.332	32	2

sapiens, ELBA on 392 nodes on Summit takes 17 minutes, while Mifiasm on Cori Haswell takes 1 hour and 30 minutes with 32 threads and Canu ran out of time after more than 64 hours of runtime with 32 threads on Cori Haswell. However, these times are from different architectures and are not directly comparable, so we do not give a speedup for H. sapiens. The results from O. sativa and C. elegans show that performance on Summit was worse than on Cori Haswell due to the less powerful processor on Summit.

Finally, we compare the quality of the assemblies of ELBA with that of Hifiasm and HiCanu. Table 5.3 summarizes metrics obtained from QUAST [87]. Specifically, we show the completeness, i.e., the proportion of the reference genome that was covered by at least one contig (the higher, the better), then the length of the longest contig (the higher, the better), the size of the contig set (the lower, the better if associated with high completeness), and the number of misassemblies (the lower, the better). In both species, ELBA has competitive genome completeness, especially in C. elegans, where it is higher than both Hifiasm and HiCanu, and misassemblies. In ELBA, the contigs are significantly shorter than in the two competing software packages. This is understandable, since ELBA does not currently perform a polishing step, which is reserved as future work.

5.5 Summary

Recent advances in sequencing technologies have increased the need for high-performance approaches, as we can now generate more data at lower cost, which in turn requires higher computational resources to reconstruct high-quality assemblies in a timely manner. In this chapter, we presented the contig generation phase of the distributed-memory long-read assembler ELBA.

Contig generation is critical to assembly functionality, as it enables the construction of longer genomic sequences that represent a physical map of a region of a chromosome. ELBA has achieved speed increases of up to 36× and 159× compared to two state-of-the-art software for the O. sativa dataset, opening the door for high-performance genome assembly.

ELBA and its contig generation step rely on distributed sparse matrices to first determine the contig set starting from a string graph. Then, using a greedy multiway number partitioning algorithm, it determines how the rows and columns of the sparse matrix representing DNA sequences are redistributed between processes so that sequences belonging to the same contig are stored locally on the same processor. ELBA then uses such partitioning and the sparse matrix abstraction to implement the induced subgraph function and redistribute the sequences among the processes. Finally, ELBA's contig generation step computes a local assembly step, i.e., the actual concatenation of sequences into a contig, on each processor independently and in parallel.

Chapter 6

High-Performance Computing in the Cloud

In the previous chapters, we demonstrated high performance and near-linear scaling for an end-to-end genomic application without sacrificing productivity using sparse matrix abstraction. The ability to productively write high-performance scientific code must be accompanied by broad and democratic access to large-scale parallel resources. It can be difficult to gain access to institutional HPC resources if the user is not part of a research community targeted by the institution or government.

In this chapter, we first describe the similarities and differences between cloud computing and traditional HPC. Then, we investigate the performance differences between cloud-based HPC systems and traditional HPC systems, and show that the cloud is now competitive with HPC systems primarily due to advances in networking technologies. Our work opens the door to a paradigm shift in high performance computing for science and makes it easier for any researcher to access large-scale parallel resources.

6.1 Overview and Foundation

The benefit of high-performance computing for scientific research has grown rapidly, beyond traditional simulation problems to data analysis in light sources, cosmology, genomics, particle physics, and more [162, 3]. Given the vast amounts of data and/or computation involved in such applications, they can require the full computing power and memory of high performance computing (HPC) systems. Cloud computing [64, 32, 113] is gaining popularity among scientists as an alternative to HPC for a wide range of sciences such as physics, bioinformatics, cosmology, and climate research [60, 117]. There are many efforts in the literature to measure the performance of scientific applications in the cloud. The lack of a low-latency network has been consistently identified as the main bottleneck [86, 125, 161, 58, 57]. Understanding the gap between HPC and cloud systems and whether the results from the literature are still valid today is critical for guiding future system design and running scientific applications efficiently in the cloud. Furthermore, the number of users that super-

computing facilities can support is limited, and the deployment of any new supercomputer is a multi-year multi-million dollar investment. Closing the performance gap between HPC and cloud would give many more scientists access to adequate computing resources.

Our work measures the performance of HPC-oriented codes on both cloud and HPC platforms. Building on previous literature, we investigate whether the findings apply to today's cloud platforms and isolate the contribution of different variables to the HPC cloud performance gap. Our results show that cloud platforms with similar processors and networks can achieve HPC-competitive performance, not only for compute-intensive applications, but also for communication-intensive applications. At moderate scales, modern cloud computing has overcome one of its main limitations by providing higher-speed memory and interconnects for HPC-oriented instances. Cost models, job wait times, software availability are also relevant to evaluating HPC-cloud competitiveness, but are out of scope here.

High Performance Computing (HPC) and cloud computing differ in their original purpose as well as their economic objectives and access policies. HPC systems were designed to deliver high performance for dedicated scientific computing, while cloud computing made networked hardware and software available for general use.

The differences in their economic objectives and access policies inevitably affect scheduling, hardware selection, and software configuration decisions. HPC systems are typically operated by a non-profit organization (university or national laboratory), funded by a government agency, and allocated to a particular research community. These systems have very high utilization (over 90%) with non-trivial wait times for users; they support very large-scale computations with homogeneous hardware that undergoes major upgrades every few years. By homogeneous hardware, we mean here that an HPC system typically has the same node for the entire machine, where that node can have both CPU and GPU devices, but the type, such as processor model and vendor, is the same for the entire system; it is possible to have two different partitions within the same HPC system, but usually no more than that. In contrast, cloud systems are built for profit, configured to meet market demand, and operated at lower utilization rates to ensure little or no wait time. Cloud resources are upgraded continuously and incrementally, leading to rapid access to new technologies, but also to heterogeneity within the cloud.

Researchers running scientific applications in the cloud can access instances with low-latency networks to achieve performance competitive to HPC [90]. However, cloud heterogeneity can limit what is available within an HPC cloud offering, e.g., it may be more difficult for a user to obtain a large number of high-performance instances. In the cloud, users can easily customize their environment without administrative overhead and quickly provision additional resources to solve large problems [161]. HPC platforms offer limited support for on-demand self-service [161], but they do offer important features such as resource pooling and broad network access.

The basic business model differences between cloud and HPC persist today and lead to complex cost trade-offs that are beyond the scope of this paper. However, the growing commercial interest in problems such as large-scale machine learning training have led to changes in cloud configurations. This has increased the popularity of HPC-as-a-Service in

the cloud and has in turn resurfaced questions about use of the cloud for modest scale parallel scientific applications.

6.2 Proposed Methodology

Processor, memory, network, application and programming model, and system age are all variables that affect performance. Here, we measure the performance gap by isolating the contribution of the different variables by dividing our experiments into two categories: (i) hardware and system and (ii) user application.

First, we isolate the contribution of processor and memory to identify similarities or significant differences in in-node performance. Then, we investigate the contribution of the inter-processor network by measuring the latency and bandwidth of communication primitives between machines. Finally, we study performance of HPC and cloud computing from an application perspective. Unless differently noted, the results reported in this paper represent the average value across 10 runs.

To this end, we use two metrics to characterize our applications: hardware events and the communication to computation ratio (Cm/Cp). The Cm/Cp ratio is defined as communication time divided by computation time for a given execution of a parallel application on a given parallel machine with explicit communication [45]. Both metrics can help interpret the potential performance gap between HPC and cloud systems.

A Hardware and System View

A common approach to comparing the performance of computer systems is to use low-level benchmarks [154, 2]. Here, we focus on the investigation of processor, memory, and network performance.

Processor. Considering a multi-core processor, we refer to it as a *node*, where a *core* is the basic execution unit in the system. The number of nodes is later denoted P. Here we use the shared memory version of the LINPACK benchmark [52] to compare the floating point performance of the systems under consideration. LINPACK measures how fast a computer solves a dense n-by-n system of linear equations $Ax = b$, which is a common task in engineering. The latest version of the benchmark is used to create the TOP500 list, which lists the most powerful supercomputers in the world [151]. The goal is to get an approximate idea of how fast a computer will perform when solving real problems. It is a simplification because no single computational task can reflect the overall performance of a computer system. Nevertheless, LINPACK benchmark performance can be a good correction to the manufacturer's stated peak performance. Peak performance is the maximum theoretical performance a computer can achieve. It is calculated by multiplying the frequency of the computer in cycles per second times the number of operations per cycle it can perform. The actual performance will always be lower than the peak performance [52]. Computer performance is a

complex matter that depends on many interrelated variables. The performance measured by the LINPACK benchmark consists of the number of 64-bit floating-point operations, generally additions and multiplications, that a computer can perform per second, also known as FLOPS. However, the performance of a computer when running actual applications is likely to fall far short of the maximum performance it achieves when running the corresponding LINPACK benchmark.

Memory Hierarchy. CacheBench [116] measures the performance of the local memory hierarchy. It computes a number of operations – *read, write, read/modify/write*, *memset*, and *memcpy* – varying the underlying array size, thereby revealing the performance of the cache. Operations run for 2 seconds and the average bandwidth (MB/s) is reported. Here we focus on *memcpy*.

Memory Bandwidth. To measure the maximum memory bandwidth of our systems, we use the STREAM benchmark [111], which performs four vector operations: *copy*, *scale*, *sum*, and *triad*. STREAM requires that (a) each array is at least four times the size of the cache memory, and (b) the size is such that the "timing calibration" output by the program is at least 20 clock ticks. STREAM provides the best possible memory system bandwidth.

Inter-Node Communication. Following standard practice [90], we use a subset of MPI operations to measure the inter-node communication performance of our systems. Specifically, we use `MPI_Send- recv` and `MPI_Alltoall` to measure point-to-point and collective latency and bandwidth using the OSU microbenchmarks [132].

A User-Application View

Besides comparing HPC and cloud systems on a subset of MPI collectives, we select two representative user applications from scientific computing as benchmarks: an N-Body simulation written in `C++` and a Fast Fourier Transform (FFT), written in `C`. N-Body is a computationally intensive application, while the FFT is more communication intensive [159, 6, 42].

An N-Body simulation models a dynamic system of particles, usually under the influence of physical forces, such as gravity [94]. It is a common computation in physics, astronomy, and biology. The naive solution computes the forces acting on the particles by iterating through each pair of particles, resulting in a complexity of $O(n^2)$, where n is the number of particles. In our implementation we consider the density of the particles to be sufficiently low so that a linear time solution can be achieved with n particles.

The FFT calculates the discrete Fourier transform (DFT) of a sequence or its inverse (IDFT). In Fourier analysis, a signal is transformed from its original domain (often time or space) to a frequency domain representation and vice versa. As a benchmark for the FFT,

Table 6.1: Details of the evaluated machines: name, system age in years, number of cores per node, processor frequency, theoretical peak performance (GFlops/s) per node, processor, memory, advertised injection bandwidth (Gigabits/s), and caches sizes. †Custom model for Amazon AWS. KNL's L2 is shared between two cores. *Advertised user-process injection bandwidth [44].

Platform	Age	Core/Node	Frequency (GHz)	Peak (GFlops/Node)	Processor	Memory (GiB)	Bandwidth (Gbps)	L1	L2	L3
Cori Haswell	4	32	2.3	1,177	Xeon E5-2698V3	120	*82	64KB	256KB	40MB
Cori KNL	4	68	1.4	3,046	Xeon Phi 7250	90	*82	64KB	1MB	-
AWS r5dn.16xlarge	1	32	2.5	2,560	Xeon Platinum 8259CL	512	75	64KB	1MB	36MB
AWS c5.18xlarge	1	36	3.0	3,456	Xeon Platinum 8124M†	144	25	64KB	1MB	25MB

we use the implementation of Frigo and Johnson [67, 66, 65], Fast Fourier Transform in the West (FFTW).

6.3 Experimental Setup

Our experiments are conducted on the Intel Xeon "Haswell" (Cori Haswell) and Intel Xeon Phi "Knight's Landing" (KNL) partitions (Cori KNL) of the Cori Cray XC40 HPC system at NERSC, an Amazon Web Services (AWS) commodity cluster with r5dn.16xlarge (R5) instances (optimized for memory-intensive workloads), and one with AWS c5.18xlarge (C5) instances (optimized for compute- intensive workloads). Details for each instance are listed in Table 6.1.

We chose these four platforms because of their easy availability and the diversity of architectures. In particular, we selected the two AWS instances to represent two extremes of the AWS catalog (memory-optimized versus compute-optimized) and selected these two instances because they allowed us to allocate multiple nodes in the same placement group in a reasonable amount of time. AWS clusters run as *dedicated instances* to reduce the potential performance slowdown from sharing resources, and use Slurm as the workload manager [163]. The need for tools to simplify the use of cloud environments and better software stacks for clouds has been noted in past literature [161]. We use AWS ParallelCluster to provision and manage AWS clusters. It automatically sets up the required compute resources and shared file system in about five to ten minutes in our experience. AWS also provides a collection of Amazon Machine Images (AMIs) installed with libraries and software such as MPI, BLAS, and TensorFlow. Notably, AWS ParallelCluster provides support for several schedulers, such as SGE and Torque (which will both be discontinued at the end of 2021), as well as Slurm and the in-house AWS Batch, which currently has limited support for GPU jobs. We used Slurm for consistency across AWS and Cori systems.

Cori Haswell and KNL also use Slurm as workload manager. Cori has the Cray Aries "Dragonfly" topology for its interconnect [124]. AWS does not disclose details about the underlying interconnect topology, except for an expected injection bandwidth (Table 6.1). The AWS cluster instances belong to the same placement group; the login node and the

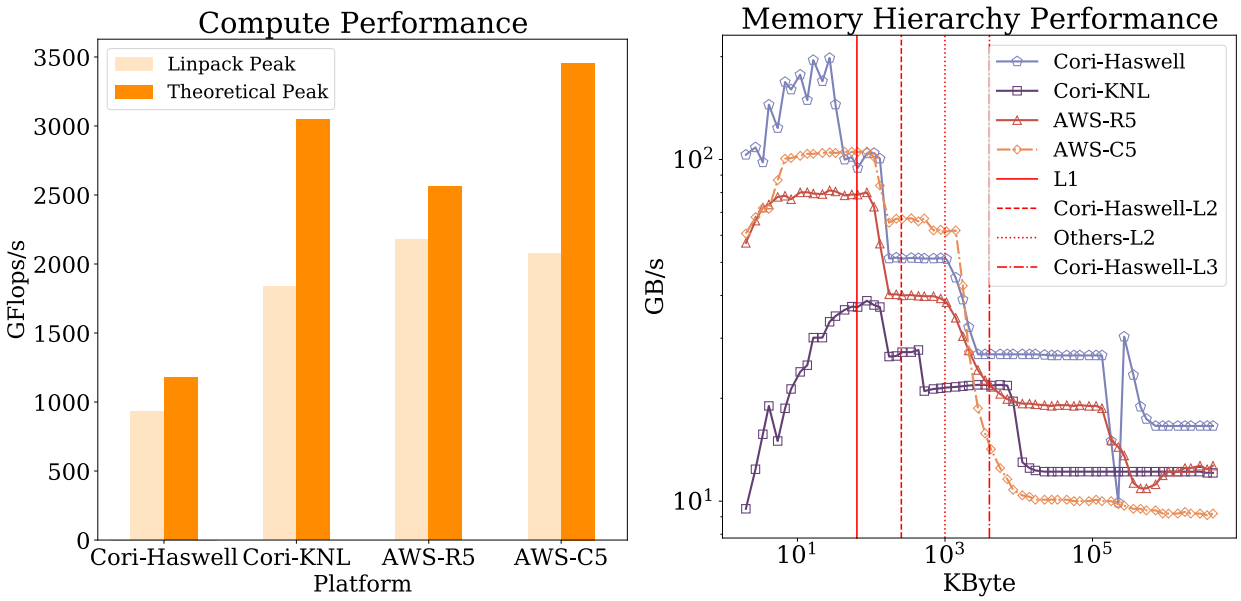

Figure 6.1: LINPACK peak performance compared to the theoretical peak (left) using one node and all available cores per node (Table 6.1). CacheBench `memcpy()` benchmark (right) using a single core and reporting the median of 10 runs.

compute nodes belong to two different subnets. A subnet is a logically visible subdivision of an IP network. The subnetwork of compute nodes is private and has no access to the Internet.

6.4 Results

A Hardware and System View

Given different in-node configurations, we first investigate performance using a microbenchmarking approach.

Processor. Figure 6.1 compares the LINPACK peak performance (left) with the theoretical peak performance (right) for each platform and also allows cross-platform comparison. Cori Haswell and AWS R5 achieve peak performance significantly closer to their theoretical peak than the other two machines. Closing the gap between theoretical peak and LINPACK peak on Cori KNL is notoriously difficult; achieving such progress requires a significant optimization effort for applications in general [15, 50, 51]. Cori KNL achieves about 350 GFlops/second more in our benchmark than the number reported in the Top500 [153]. This discrepancy could be due to different implementations of the LINPACK benchmark, since we use the Intel Math Kernel Library benchmark package. Further profiling of AWS C5 with

Table 6.2: STREAM benchmark: as many OpenMP threads as the number of physical cores per node (top) and one thread (bottom), 8 bytes per array element, array size = 120000000 (elements), offset = 0 (elements), memory per array = 915.5 MiB, total memory required = 2746.6 MiB. The best time for each kernel over 10 runs (excluding the first iteration) is used to compute the bandwidth. Results in GB/s [111].

Platform	**Threads**	**Copy**	**Scale**	**Add**	**Triad**
Cori Haswell	32	56.6	43.6	49.4	49.7
Cori KNL	64	247.9	250.3	257.1	260.0
AWS r5dn.16xlarge	32	181.9	127.6	143.9	144.9
AWS c5.18xlarge	36	135.7	106.9	120.4	120.3
Cori Haswell	1	18.0	11.3	12.6	12.6
Cori KNL	1	12.1	6.8	8.4	7.4
AWS r5dn.16xlarge	1	11.1	12.5	13.2	13.1
AWS c5.18xlarge	1	11.0	12.6	13.5	13.6

VTune [139] revealed a relatively low core utilization for this platform, which could explain the large gap between theoretical and achieved peak.

The cloud instances perform best in absolute terms. AWS R5 and C5 instances are equipped with newer hardware than Cori systems; this may explain the greater processing power. It is noteworthy that the elastic nature of cloud computing – as opposed to multi-year projects to develop and install supercomputers – offers the potential for rapid hardware turnaround.

Memory Hierarchy. Figure 6.1 shows the results for the Cache-Bench benchmark (on the right) and illustrates the performance of the cache hierarchy for our four machines. For each platform and size, we ran the benchmark 10 times and report the median; there is little variance among different runs for a given size and platform.

Cori Haswell has the best performance for L1 (which is the same size on all machines). The L2 performance of Cori Haswell and AWS C5 are comparable, while the performance of Cori Haswell falls below that of the AWS C5 platform below its second cache level. AWS C5 achieves better performance than Cori Haswell as long as the data fits into its L2 cache, and its performance falls below Cori Haswell when it enters the third cache level as expected because Cori Haswell has a larger L3 cache.

Considering the data in Table 6.1, one would expect a higher bandwidth for AWS R5 and Cori KNL given their larger L2. However, the way the caches are shared between the cores and cache associativity could affect overall memory throughput. For Cori Haswell, L2 is private to each core, while for Cori KNL it is shared by two cores. Cori KNL has two cache levels instead of three like the other machines. Cori Haswell's cache is 8-way associative, while Cori KNL has a direct mapped cache. This direct mapping reduces cache management

complexity, but can significantly increase cache thrashing, resulting in a high rate of cache misses and main memory accesses [123].

Looking only at these single core results, one might suspect that the virtualization overhead could prevent cloud instances from fully exploiting the potential of their caches. However, our results, which measure the performance of the whole memory system, discourage this hypothesis, as shown in the next microbenchmark.

Memory Bandwidth. To measure memory bandwidth when data does not fit in the system cache, we run the STREAM benchmark [111]. The results in Table 6.2 show that the performance difference between Cori Haswell and AWS R5 and C5 (Figure 6.1) is reversed in favor of the cloud clusters when all available cores are used if the data does not fit in the platforms' caches.

Cori KNL has the higher memory bandwidth thanks to its on-chip multi-channel DRAM (MCDRAM) chip of 16GB. Looking at platforms without on-chip memory, cloud instances show a significantly higher memory bandwidth than the corresponding HPC platform. System age and newer cloud hardware can explain this performance. These results suggest that a faster hardware turnaround time could benefit not only computationally intensive applications, but also data-intensive applications. In addition, these results discredit the hypothesis that virtualization overhead is a major limitation of today's cloud computing.

Inter-Node Communication. To study network performance, we measure bandwidth and latency in a multinode setting. In our experiments, we use `openmpi-4.0.2` as the MPI implementation. For Cori Haswell and KNL, we ran the benchmark suite with both `openmpi-4.0.2` and the default `cray-mpi`. They provided similar performance, and we decided to report only the results for OpenMPI for clarity and consistency with the cloud instances.

Figure 6.2 uses one process per node to show point-to-point bandwidth (left) and latency (right). Our results show peak bandwidth of about 86 Gbit/s for AWS R5 and 90 Gbit/s for AWS C5, while Cori Haswell and Cori KNL show peak bandwidths of 74 and 64 Gbit/s, respectively. Considering that the two Cori systems share the same network, one would expect the same network performance, however, their performance in Figure 6.2 are significantly different. This difference can be attributed to the overhead of MPI function calls, which are expensive and penalise lower frequency Cori KNL cores that cannot match the performance of Cori Haswell nodes. Our results are consistent with those presented by GASNet [68].

Cloud instances outperform HPC systems in both bandwidth and latency. Until recently, the lack of a low-latency network has been consistently identified as the main bottleneck of cloud computing for scientific applications [161, 86, 125]. Our results show that modern cloud computing has made significant advances in networking technology that provide cloud instances with HPC-competitive network performance.

Figure 6.3 shows the `MPI_Alltoall` latency on P=2, 8. For small message sizes, Cori Haswell dominates the other platforms on two nodes; the gap decreases as the number of

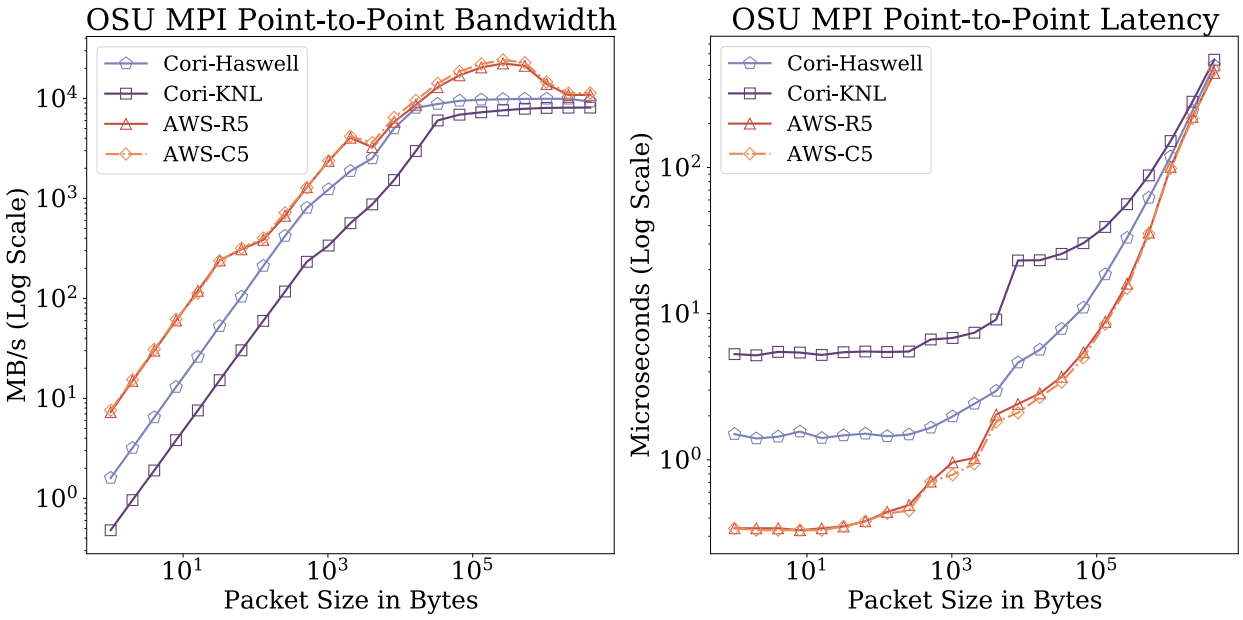

Figure 6.2: OSU MPI microbenchmark injection bandwidth (left) and point-to-point latency (right) in log-log scale. Using two nodes with one process per node on Cori Haswell, Cori KNL, AWS R5, and AWS C5.

nodes is increased. The differences between Cori Haswell and KNL are due to the cost of MPI calls on the two different processors. Also, the different number of processes per node in this experiment illustrates the difference between the two Cori systems. On two nodes, the gap decreases as the message size increases, especially when comparing Cori Haswell and AWS R5, whose performance almost overlaps at large message sizes. AWS R5 shows similar performance to Cori Haswell on eight nodes, except for small message sizes. Looking only at the historical results, one would expect the cloud instances to lose performance and the gap to grow as the number of nodes increases. On the contrary, our results show significant improvements, so one can expect better performance scaling as the number of nodes increases. AWS R5 performs as we would expect given its performance in the previous microbenchmark, while AWS C5 is far from Cori Haswell. Its advertised network bandwidth is about 3× lower than Cori Haswell and since it is a compute-optimized instance, we suspect it may suffer from network contention.

Our results suggest that the place we would expect HPC to retain an advantage is in applications with many small messages. Algorithmic techniques, however, typically try to avoid this situation. These results have important implications for communication-intensive applications that have not historically benefited from cloud computing due to their bandwidth requirements.

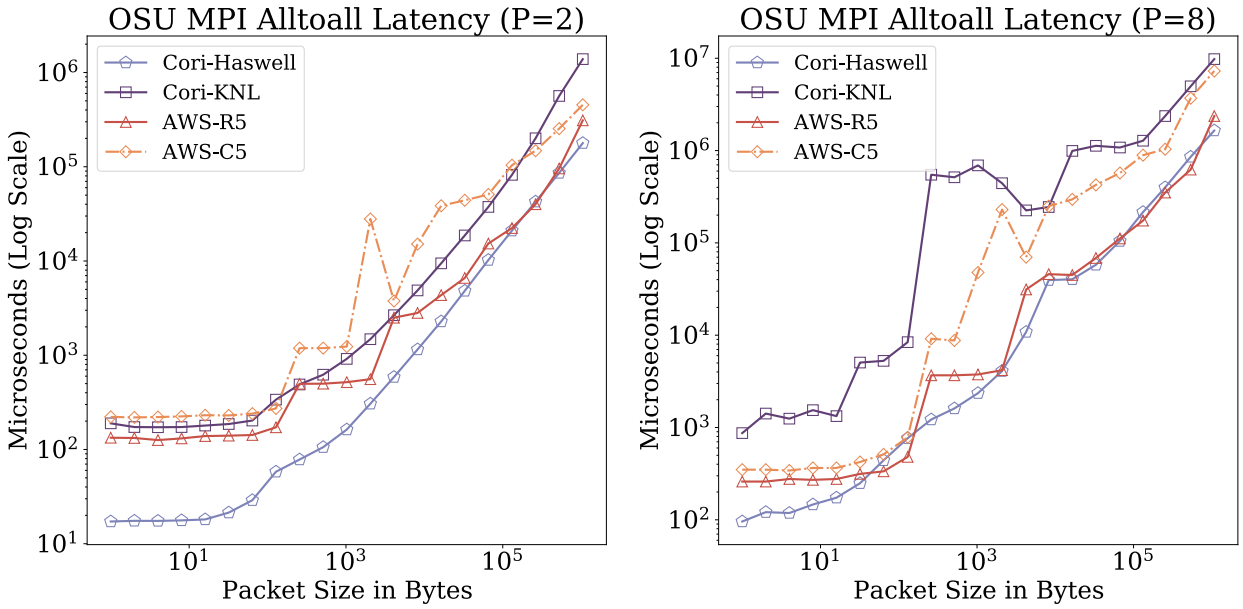

Figure 6.3: OSU MPI microbenchmark `MPI_Alltoall` latency on two nodes (left) and on eight nodes (right) in log-log scale. On Cori Haswell, AWS R5 and C5 we use 32 processes per node, while on Cori KNL we use 64 processes per node.

Table 6.3: Characterization of n-body using `perf` run on a single core. Page size = 4KB, problem size: 1M.

Platform	Instruction (G)	Page Fault (K)	Cache Miss (M)	Time (s)
Cori Haswell	414.7	367.2	11,347.8	461.7
Cori KNL	415.4	367.4	11,220.1	1,736.5
AWS r5dn.16xlarge	-	367.2	-	486.9
AWS c5.18xlarge	427.2	367.2	21,457.4	480.6

An Application View

In this section, we first measure and compare the serial runtime of the applications and analyze the single-core performance of the applications to better understand the runtime differences and similarities between the machines. Then, we study the parallel performance of the applications in a multinode environment.

Serial Performance

In Tables 6.3 and 6.4, we report the single core performance for the N-Body simulation and the FFT, respectively. In both applications, Cori KNL has a significantly higher runtime than the other machines. Its poor performance can be justified by the lower frequency of

Table 6.4: Characterization of FFT using `perf` run on a single core. Page size = 4KB, problem size: 50K.

Platform	Instruction (G)	Page Fault (K)	Cache Miss (M)	Time (s)
Cori Haswell	782.1	9,766.8	871.5	312.4
Cori KNL	784.9	9,766.8	20,915.0	2,348.1
AWS r5dn.16xlarge	-	9,767.5	-	303.3
AWS c5.18xlarge	1,097.9	9,766.6	2,953.6	335.8

its processor and the poor performance of its memory system. Cori KNL's clock speed is about half that of the other cores in the study, and it needs all 68 of them to compete with the (theoretical) GFlop rate of the other 32-36 core nodes. Recall that the L2 caches on Cori KNL are shared by two cores, while they are private on the other machines. In fact, the performance for FFT is relatively worse since it is a more memory intensive application than N-Body. Cori Haswell and the two cloud instances show similar runtime for both applications. Cloud instances have lower cache performance than Cori Haswell, while they have higher bandwidth when data can no longer fit in the cache. Since we study single-core performance here, the lower half of Table 6.2 shows that Cori Haswell and the AWS instances have comparable performance in the single-core STREAM benchmark.

Overall, these results are consistent with the results of our microbenchmarks and confirm that cloud virtualization overhead has decreased to a point where application performance is not significantly impacted. As a result, cloud instances have comparable runtime to a HPC system for both applications.

Workload Characterization

Recall, when we measure the runtime of an application, we measure both the processor and the memory system. Runtime alone is not enough to get a reasonable understanding of the variables that affect application performance.

Here, we extend our analysis by measuring the number of *page faults*, *instructions* and *cache misses* for each application on each platform and comparing the results. A high rate of page swapping-in/out, cache misses, and a high number of instructions can significantly slow down applications [146, 11, 101]. On all systems, these metrics are measured for a process on a single node using `perf` [157]. Cache misses and instructions are not available for AWS R5. In particular, it is not easy to get access to accurate hardware counters. On HPC systems they typically require administrative privileges, while on cloud systems it can be difficult to separate the effects of virtualization and gain access to accurate metrics.

Tables 6.3–6.4 give the number of page faults on the machines for the N-Body simulation and the FFT. The number of page faults is mostly the same and confirms the same behavior across the four machines. Cori Haswell and Cori KNL automatically load a software package to increase the page size from 4K to 2M. This setting was unloaded and disabled to allow a fair comparison between the four machines. Similarly, the page size could have been increased

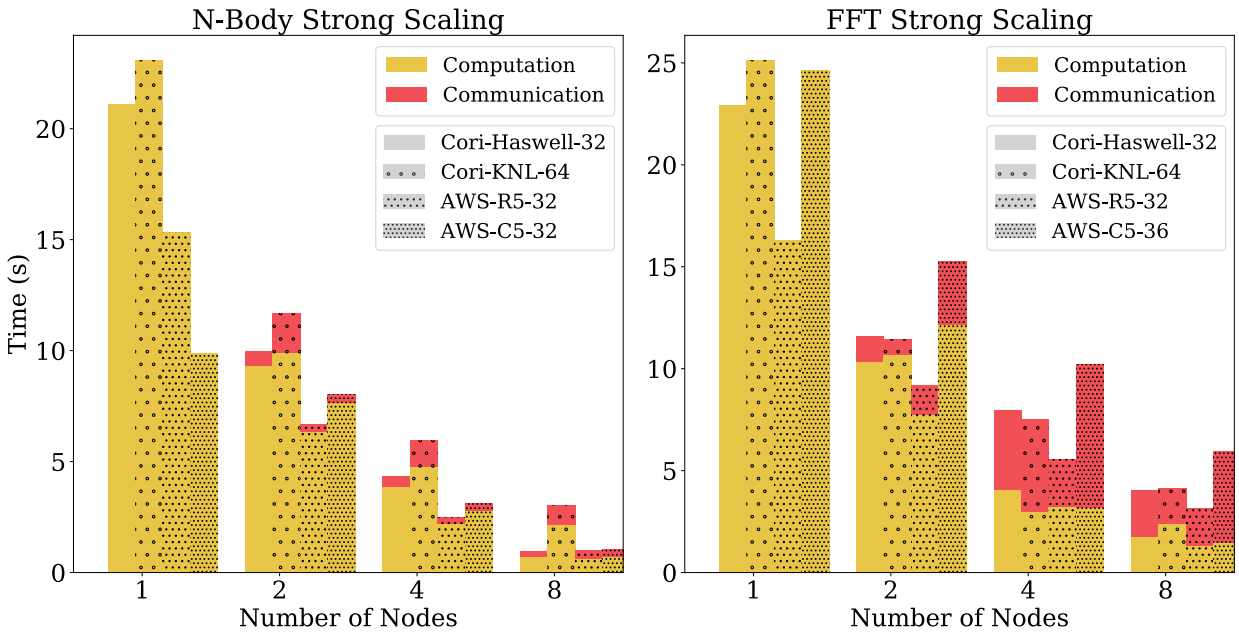

Figure 6.4: The N-Body strong scaling with 1M particles (left) and the FFT strong scaling with 50K points (right) across the machines. The number next to the name in the legend indicates the number of processes per node.

on the AWS instances. For simplicity, we chose to reduce the page size on Cori and do not expect this setting to change the overall trend of our results. The only significant difference in the number of instructions is between the Cori systems and AWS C5 for the FFT. This difference could explain the runtime difference between Cori Haswell and AWS C5, although it is not large.

Cache misses show a more relevant impact on performance than page faults and instructions. The Cori systems have similar cache misses for N-Body simulation, while they show a significant gap for FFT. Cori KNL's direct mapped cache significantly penalizes its performance for a memory-intensive application such as FFT. AWS C5 has a larger number of cache misses than Cori Haswell for both applications. This result, combined with AWS C5's slower L1 (Figure 6.1), suggests that cache misses are one of the variables contributing to the runtime difference between these two machines.

Our workload characterization reveals that cache misses and memory system performance have the largest impact on single-core performance. Nevertheless, the resulting runtime differences are small, and our analysis shows comparable single-core performance between Cori Haswell and the cloud instances.

Parallel Performance

In examining parallel performance to highlight the effect of the network, we report the median of 10 runs of the application for P=1, 2, 4, 8. Due to a limit on the number of instances we can create simultaneously, we were unable to get more than eight instances in the same placement group, which is critical for achieving low-latency network performance. AWS support can increase this limit upon request. Given the varying number of cores per node of our machines, we normalize our results and specify the configuration that provides the best performance for each platform.

N-Body Simulation. Figure 6.4 on the left illustrates the strong scaling performance across the machine and shows the runtime split in computation and communication. Our N-Body implementation uses a recursive doubling algorithm for particle exchange and therefore runs much faster with the power of two processes. As a result, all machines achieve their best performance with either 32 or 64 processes per node.

The N-Body simulation is computationally intensive and has a low Cm/Cp ratio, suggesting a modest impact of the network on overall runtime. Given the Cm/Cp ratio and the serial performance of this application, we expect comparable runtimes between Cori Haswell and the two cloud instances. Cori KNL also has comparable runtimes, while its P=8 scaling is significantly worse than the other three machines. In particular, it uses twice as many processes per node as the other machines and runs at about half the frequency, with fewer cache levels and L2 caches shared by two cores.

The `MPI_Alltoall` microbenchmark shows a significant difference between the two Cori systems. Remember that the Cori systems use the same network; however, Cori KNL uses 64 processes per node instead of 32 and the MPI function calls overload the weaker KNL cores. The gap is also significant between Cori Haswell and AWS C5 at any scale, while the gap between Cori Haswell and AWS R5 is mostly overlapping.

The `MPI_Alltoall` gap between the two Cori systems is reflected in their performance in the N-Body simulation. Therefore, one would expect AWS C5 to have a larger communication time. Nonetheless, the performance of AWS C5 is consistent with the assumption that when the Cm/Cp ratio is low, the network has a limited impact on the overall application runtime.

Overall, AWS R5 is the fastest platform, with Cori Haswell and AWS C5 having the same performance at P=8. The remarkable comeback of Cori Haswell might be due to fitting data into the larger L3 cache of Cori Haswell. The N-Body simulation scales superlinearly – on all machines except Cori KNL. We are familiar with this implementation and know its superscaling behavior, which can be briefly explained as the "cache effect", meaning that as the number of nodes increases, more data fits into the cache.

Our results confirm that cloud computing can be more suitable than HPC systems for computationally intensive applications [161] and that modern cloud computing can provide competitive network performance to HPC.

Fast Fourier Transform. Figure 6.4 shows the strong scaling performance of the FFT (right) and splits the runtime into computation and communication. The library FFTW computes multiple FFTs and measures their execution times to find the optimal plan that achieves the best performance for each machine. We use these optimal implementations. Since the optimal plan selected by FFTW is based on a collection of `MPI_Sendrecv`s, the results in Figure 6.2 are relevant to the following analysis.

FFT has a higher Cm/Cp ratio than the N-Body simulation, and as expected, Figure 6.4 shows that the communication overhead is much higher than in the previous application and can take more than 50% of execution time. There is a consistent spike in communication at P=4, which we suspect is due to implementation details of FFTW. On all machines, the overall scaling of the FFT is sublinear, mainly due to communication overhead. AWS R5 is the fastest platform, both in terms of total and communication time. It is followed by Cori Haswell. The computation times of Cori KNL and AWS C5 are comparable, but AWS C5 has a higher communication overhead, making it the slowest platform in this benchmark.

Despite comparable performance for point-to-point communication (Figure 6.2), the cloud instances exhibit different performance for all processes on the node involved in the communication (Figure 6.3). AWS C5 exhibits significantly worse performance for the `MPI_Alltoall` benchmark, which explains the difference in communication performance between the two AWS instances for the FFT results (Figure 6.4). AWS R5 is optimized for memory-intensive workloads, while AWS C5 is optimized for compute-intensive workloads. Moreover, AWS C5 uses Amazon's in-house EFA interconnect, whose advertised bandwidth is 3× lower than R5's. Our hypothesis is that as the number of processes increases, the C5 interconnect is more subject to contention than R5's network.

AWS R5 is the best performing platform in this benchmark, as one would expect based on the results of our microbenchmarks and workload characterization. The communication time on AWS R5 is comparable to or even lower than that on HPC systems. Thus, it is not only the newer processor that contributes to the high performance for this application, but also the interconnect speed. Previous literature has shown that FFTs for cloud instances have significantly lower performance than for HPC systems. The Magellan report [161] describes the FFT as 4 to 20× slower than the HPC systems considered, running on 8 processes per node and P=8. Our result is an important validation of the recent advances that cloud computing has made in networking technology to close the performance gap with HPC.

6.5 Summary

Our work investigated the performance gap between current HPC and cloud computing systems to understand the nature of their differences and guide the design of future cloud systems. In this work, we analyzed the cross-stack performance, from single core compute power, to memory subsystem, inter-node communication performance, and overall application performance.

In particular, we highlight that cloud computing can offer a greater variety of hardware configurations and newer technology due to continuous procurement cycles. If a study requires the latest technology or a particular memory size and processor type, these are more likely to be available in the cloud, while a given HPC system may offer only one or a small set of standardized resources suitable for typical scientific applications. Our results contradict earlier findings on cloud interconnects, namely that networks for HPC instances within the cloud have improved to the point of providing competitive performance to that of HPC systems at modest scales.

On the other hand, cloud policies can limit what is available within an HPC cloud offering, e.g., one may need to make a request to the vendor to use more than a few instances, and the latest node architectures may not be available with the fast network. In contrast, in traditional HPC systems, the entire system typically has the same network, whose performance is mostly determined by the age of the system, as the procurement cycles are typically longer.

Our results showed that the compute and memory subsystem performance of cloud instances is competitive with HPC systems. This is consistent with historical results demonstrating cloud competitiveness for compute-dominated workloads.

Cloud systems offered higher bandwidth and lower latency than HPC systems for point-to-point communication. In the FFT benchmark, which is bisection-bandwidth limited, the performance of the compute-optimized cloud platform dropped, possibly due to network contention, while the platform optimized for memory-intensive applications significantly outperformed all other machines. This represents a significant advance in cloud computing technology, as the performance of multinode FFT applications on HPC systems has historically been better than on cloud systems [161]. A larger scale performance study focusing on machine balance would be an interesting future work to analyze the gap on a larger scale.

Our work shows that today's cloud computing can provide competitive performance to HPC, not only for compute-intensive applications, but also on memory- and communication-intensive workloads. The recent performance improvements of cloud instances may be due to the increasing demands of deep learning [97, 89], potentially benefiting seemingly unrelated computational science as a byproduct. It is worth noting that our study focused on one cloud provider and it would be important to replicate the study on other providers to draw more generalized conclusions. Given our results, an important future work would be a comparison focusing on elasticity and resource management, which together with our results would allow users to make informed decisions about which system is better suited for their applications.

This work has attracted the interest of other cloud companies (Google, IBM, Microsoft, and Amazon) who have provided financial support to extend our work. A preliminary performance study of Google Cloud Platform (GCP) would support the statement that the cloud as a general entity, not just Amazon Web Services, is opening the door to a paradigm shift in high-performance computing and scientific computing.

Chapter 7

Related Work

The paradigm for assembling long-read sequencing data is composed of three main stages: finding overlapping sequences to create an overlap graph, removing redundant information to simplify the graph, and creating the contig set. In this chapter, we summarize the state of the art for these three main stages of *de novo* long read assembly pipelines.

7.1 Overlap Detection

Overlap detection is the first and most computationally intensive step of the assembly paradigm, whose goal is to find overlapping areas between sequences. Overlap detection is a common step in many pipelines, not only in genome assembly, but also in error correction, for example. Here we focus on tools performing overlap detection and mention whether they are associated with or integrated into an assembly tool.

DALIGNER [120] uses k-mers to find overlap candidates and then perform alignment. It parses the sequences in k-mers, sorts them, and finds overlapping sequences with a merge operation. To filter out spurious overlap candidates, a pairwise alignment is performed using a linear expected-time heuristic based on the difference algorithm [118]. DALIGNER is integrated with the FALCON assembler [40] for the raw long read error correction phase.

BLASR [35], originally developed to align noisy long-read sequencing data to reference genomes, later became popular as a read-to-read aligner. It too first uses k-mers to detect initial overlap candidates and then filters them using alignment. In addition to the aligner itself, Chaisson and Tesler [35] presented a mathematical model that proved the feasibility of using a k-mer seed to find a match between a noisy long-read sequence and a correct reference sequence. In this paper, we make a similar contribution by presenting a different model that proves the feasibility of using a k-mer seed to find a match between two noisy long-read sequences, and that is not restricted to regular k-mer selection strategies. BLASR used to be integrated with FALCON-Unzip assembler [40] to align sequences to the contig set. It was then replaced by minimap2 [104, 130], which is presented next.

Li's minimap2 [103, 104] also uses seeds to find matches. However, it uses a different kind of k-mer, called a minimizer, which reduces the number of seeds because it selects

only one minimizer in a window w whose value is the minimum according to a function. It does not perform any alignment. Instead, it computes an approximate alignment score based on the location of the minimizers on the sequences and excludes those whose quality is below a defined threshold. This tool is part of the hifiasm assembler [38] as well as of FALCON-Unzip [40].

MECAT [160] identifies overlap candidates based on k-mers and introduces a pseudo-linear alignment scoring algorithm to filter out excessive candidates by using a distance difference factor to score k-mer matches. The score of the k-mer seed pair is supported by all matching k-mer pairs and their interval distance, so the scores represent the global matching information between two sequences. The score of a k-mer seed pair of a read pair increases linearly with the length of their overlap. Therefore, the algorithm chooses read pairs with high scores and removes non-informative candidate matches. MECAT is associated with the homonym assembly tool.

MHAP [16] is a probabilistic algorithm for sequence overlap detection. It estimates Jaccard similarity by compressing sequences to their representative identity using the MinHash algorithm [23] and filtering out false candidates. Once the overlap candidates are found, the overlap regions are calculated based on the relative median positions of the shared seeds. These overlaps are validated using counts of a second set of common seeds, which may be smaller within 30% of each overlap region. MHAP is associated with both Canu [100] assembler and HiCanu assembler [127].

A modified approach based on MinHash is also used in the Shasta assembler [145] for overlap detection while the Raven assembler [156] uses a combination of minimap [103] and MinHash for overlap detection.

The wtdbg2 assembler [142] first parses the sequences and counts the k-mer occurrences. Then it takes each subsequence with 256 base pairs in the reads as a unit defined as a bin, and creates a hash table whose keys are k-mers that occur twice or more in the reads, and whose values are the positions of the corresponding bins in the reads. From these bins, they can determine overlapping matches between sequences.

Simpson and Durbin [148] use the Ferragina–Manzini index (FM–index) [62] derived from the Burrows–Wheeler transform [31] for overlap detection. Bonizzoni et al. [19, 18] propose a similar approach using only the FM–index to construct a string graph.

Our first distributed memory design for overlap detection, diBELLA 1D [57], uses a k-mer-based approach and traverses a distributed hash table to find overlapping sequences. The keys of this hash table are the k-mers and the values are lists of sequence identifiers in which the k-mer occurs. This design is similar to a 1D SpGEMM that uses an outer product algorithm without explicit construction of matrices.

7.2 Transitive Reduction

The transitive reduction step is crucial to move from an overlap graph, which contains a lot of redudant information, to a string graph, which facilitates the extraction of linear sequences

of vertices, i.e., contigs.

Myers' transitive reduction algorithm consists of iterating over each node v in the source graph and examining nodes up to two edges away from v to identify all transitive edges that leave or enter v [119]. These edges are then marked for removal, and they are removed after all nodes have been considered. The Myers algorithm is the common approach used by most long read assembly softwares that follow the OLC paradigm [100, 127, 38].

Jackson and Aluru [95] present a parallel algorithm for constructing a bidirected string graph from a de Bruijn graph [112] where vertices are k-mers and edges are single nucleotides whose two vertices have in common. The De Bruijn graph is not suitable for long read data due to high error rates.

SORA [133] computes transitive reduction of a string graph based on an overlap graph in distributed memory using Apache Spark [164] and the GraphX library [76], which allows parallel computation on distributed graphs in Spark. To the best of our knowledge, SORA is the only other distributed algorithm that computes transitive reduction on overlap graphs, although it was designed for cloud environments.

7.3 Contig Generation

Contig generation is an important step in any *de novo* genome assembly pipeline, regardless of the type of sequencing technology (long-read or short-read). The goal is to extract linear sequences of vertices, i.e., sequences, from a string graph.

HiCanu [127] and Falcon-Unzip [40] implement similar approaches inspired by the Bogart algorithm presented by Canu [100] to generate the contig set in shared memory starting from a sparse long-read overlap graph or a string graph. The Bogart module creates an assembly graph using a variant of the *best overlap graph* strategy of Miller et al [115], where a *best* overlap is the longest overlap to a given read end excluding *contained* sequences (i.e., when all bases in one sequence are aligned to another sequence). The Bogart algorithm removes overlapping sequences from the overlap graph to include only those that are within some tolerance of the global median error rate, and recalculates the longest overlapping sequences using only that subset (i.e., a sparse overlap graph). Bogart generates the initial contig set from the maximum non-branching paths in that graph.

The Shasta assembler [145] also uses a similar procedure by creating an undirected graph where each vertex is an oriented read (i.e., each read contributes two vertices to the read graph, one in its original orientation and one in the reverse complement orientation) and an undirected edge is created between two vertices when we find an alignment between the corresponding oriented sequences. To reduce the high connectivity in the repeat regions, the Shasta assembler preserves only the k-nearest neighbor subset of the edges. The contig set is created from the linear structures of the graph.

In contrast, Hifiasm [38] generates a primary assembly based on the topological structures of the graph and the phasing relationship between the different haplotypes using a

bubble-popping procedure [103]. Finally, a best overlap graph is used to deal with remaining unresolved substructures in the assembly graph.

A De Bruijn graph-based approach is common [112] for genome assembly of short-read sequencing data, but it has not been suitable for long-read data in the past because of higher error rates.

PaKman [73] is a parallel distributed memory contig generation algorithm for short-read sequencing technology. PaKman introduces a new compact data representation of the De Bruijn graph, named PaK-Graph, which uses iterative compression to fit the graph into the memory available on each node. This compression enables low-cost replication of the graph across nodes, reducing the need for communication and creating an embarrassingly parallel procedure. PaKman also introduces an algorithm to perform non-redundant contig generation, i.e., to avoid two processes traversing the same path in the graph and generating the same contig.

MetaHipMer [69, 93] and the earlier HipMer [70] are distributed-memory *de novo* metagenome and genome assembly pipelines, respectively, designed for short-read data and thus also use the De Bruijn-based approach. Both are implemented using a partitioned global address space model in either UPC [33] or UPC++ [12]. Contig generation is performed after the construction of a distributed hash table of k-mers with the left and right base extension from the input data stored with each k-mer. Each process then creates contigs by starting at a k-mer, walking left and right, appending the extension, and looking up the resulting k-mer in the hash table. These lookups often take place on remote nodes and are performed with either a remote memory operation or a remote procedure call. If a previously visited k-mer is reached during this process, the two contigs are merged. Fine-grained synchronization prevents a data race that can occur when two processes attempt to merge at the same time.

As error rates decrease for long-read data, we find in the state of the art some attempt to use a De Bruijn graph approach for this type of sequencing technology, such as the shared-memory assembler Flye [99]. Instead of generating a contig set, Flye first generates a *disjointig* set, i.e. concatenating multiple disjoint genomic sequences, and then concatenates these error-prone disjointigs into a single string (in any order), constructs an assembly graph from the resulting concatenation, uses sequences to disentangle this graph, and resolves bridged repetitive areas (which are bridged by some sequences in the repeat graph). It then uses the repeat graph to resolve unbridged repetitive areas (that are not bridged by sequences) based on the differences between repeat copies. The output is a contig set generated from the paths in this graph. Using a De Brujin approach in conjunction with long read sequencing data leads to more complex algorithms than those commonly used in an OLC-based assembler. This makes an approach such as Flye's unsuitable for scalable implementation on distributed memory architectures, as the access patterns are extremely fine-grained.

7.4 Parallel Strategies for Unstructured Computation

Unstructured and irregular computation poses non-trivial challenges to the programmer when it comes to implementing it in parallel, especially when distributed memory parallelism is involved. Nonetheless, this type of computation is widely used in scientific computing and thus we find work in the literature that tries to overcome these challenges.

Partitioned Global Address Space (PGAS) [4, 72, 37] is a programming model paradigm that defines an abstraction of global memory address space that is logically partitioned, with a portion local to each process. PGAS is a distributed memory programming model paradigm that gives the programmer the appearance of writing a program for shared memory parallelism. There are many programming models that use the PGAS paradigm, such as Coarray Fortran [147], Unified Parallel C [71], Split-C [46], Chapel [131], UPC++ [12], and SHMEM [36], which often allow the programmer to use asynchronous or one-sided communication, e.g., through Remote Memory Access (RMA), making this paradigm a suitable choice for implementing irregular and unstructured computation [70, 93, 20].

Marquita Ellis' dissertation [59] describes how to efficiently manage irregular all-to-all computation and compares bulk-synchronous approaches based on collective communication (MPI-based) and asynchronous approaches based on one-sided communication (UPC++-based). The bulk-synchronous approach makes extensive use of global communication collectives that exchange data across processes in a single stage or, to save memory, in a series of irregular stages. The asynchronous approach provides lightweight Remote Procedure Call (RPC) techniques to transfer both data and computational work between processes. RPC techniques give the user the ability to call a function on remote processes rather than using low-level memory access primitives as in RMA.

Benjamin Brock's dissertation [20] explores techniques for building high-level, cross-platform distributed data structures for irregular computation using one-sided remote memory operations using the PGAS model paradigm. He also investigates RDMA-based (Remote Direct Memory Access) distributed data structures, including hash tables, queues, and dense and sparse matrices implemented in the Berkeley Container Library (BCL) [21].

GraphBLAS [98, 47] is an API specification that defines standard primitives for graph algorithms in the linear algebra language. GraphBLAS is based on the notion that a sparse matrix can be used to represent a graph as either an adjacency matrix or an incidence matrix. GraphBLAS is currently available for shared memory [27] and work is underway to extend it to distributed memory [22].

7.5 Sparse Linear Algebra in Genomics

Regarding the use of sparse matrices and matrices in general for genomic computations, there is related work for both shared and distributed memory machines. The shared-memory software BELLA [83] is the first work to propose the use of sparse matrices in the context of *de novo* genome composition, focusing on the overlap detection phase. A sparse matrix $\mathbf{A}$ is

used to indicate the presence of k-mers in sequences, and by multiplying by their transpose, i.e. $\mathbf{A}\mathbf{A}^\mathsf{T}$, BELLA identifies overlapping sequences.

diBELLA 2D [85] resembles BELLA, in that it computes both overlap detection and transitive reduction over an overlap graph as distributed SpGEMM and sparse computation. PASTIS [144], similarly, computes protein homology search as distributed SpGEMM.

Besta et al. [17] present another approach similar to BELLA using distributed SpGEMM to calculate the Jaccard similarity between read sets of different genomes. The main difference is that their software is optimized for the case where the output $|genomes|$-by-$|genomes|$ matrix is dense because it stores the Jaccard similarity between any genome pairs.

7.6 High Performance Computing in the Cloud

There are many efforts in the literature to measure the performance of scientific applications in the cloud. The lack of a low-latency network has been consistently identified as the main bottleneck [86, 125, 161, 58, 57].

These studies have shown that the cloud delivers competitive performance for HPC applications with minimal communication and I/O, but significantly underperforms for memory- and communication-intensive workloads. Virtualization overhead has also been identified as a performance-limiting factor, but studies do not generally agree on its impact. He et al. [90] (2010) concluded that virtualization technology has no significant performance overhead, while the results in the Magellan report [161] (2011) and by Gupta et al. [86] (2014) show that virtualization overhead along with slow network is one of the major limitations of the cloud. Performance variability due to resource sharing and the lack of tools for using and managing cloud environments–such as batch scheduling and base images–have also further limited the competitiveness of the cloud for scientific computing [161, 86, 125].

It is important to note that in 2022, after the publication of our work in 2021, Reed, Gannon, and Dongarra come to the same conclusion that the major cloud companies have invested in massive-scale systems that dwarf today's HPC systems [137]. Driven by the computational demand of AI, these cloud systems are increasingly being built with custom semiconductors, reducing the financial leverage of traditional computing companies. These cloud systems are now changing the way we think about the nature of scientific computing.

Chapter 8

Conclusions

In this dissertation, we presented a novel set of parallel and distributed algorithms for assembling *de novo* long-read genomes using sparse matrix computation and semiring abstraction. The sparse matrix abstraction not only enabled better parallelization strategies and thus better performance and higher productivity, but also improved algorithm flexibility and modularity, which is critical for genomics software packages that need to adapt quickly to new sequencing technologies. Our algorithms achieve high parallel efficiency on thousands of cores on supercomputing machines with speedup up to 159× over a popular competing assembler, reducing the runtime for human genome assembly from days to minutes while showing promising assembly quality. Our algorithms for long-read assembly are presented in a software package called ELBA (Extreme-Scale Long Read Berkeley Assembler). The code for ELBA is open source and can be downloaded at: `https://github.com/PASSIONLab/ELBA`. The input data used for evaluation by this dissertation is available for download at: `https://portal.nersc.gov/project/m1982/ELBA`.

In addition, we introduce several new mathematical and probabilistic methodologies integrated into ELBA to facilitate selection in the hyperparameter space and ensure high-quality outcomes. These methodologies prove that a seed-based approach for long-read sequencing data is feasible and help us to choose the optimal k-mer length, the boundaries of the k-mer set in terms of abundance, and the threshold for the pairwise alignment similarity score. Our methodologies are general in nature and can be applied to other k-mer and pairwise alignment strategies than the one implemented in ELBA.

This dissertation paves the way to high-performance genomics by addressing some of the computational challenges that limit the far-reaching impact of genomics on everyday life. However, developing highly parallel and scalable algorithms is not enough to democratize high-performance computing for genomics, as supercomputing machines are typically allocated to specific research communities and access to them can be difficult. Therefore, as a step toward democratic high-performance computing, we also explore alternatives to traditional supercomputing systems in this dissertation. In particular, we performed a performance comparison between today's cloud computing systems and traditional HPC systems. Our results show that cloud computing today can compete with traditional HPC systems not only for compute-intensive applications, but also for communication-intensive applica-

tions, at least at moderate scales, thanks to significant advances in networking technology. For both compute-intensive and communication-intensive applications, cloud instances with similar advertised network bandwidth to the supercomputer offered better communication performance than the supercomputer. These results mark the beginning of a paradigm shift in high-performance computing for scientific computation.

In conclusions, we believe that this dissertation will impact not only *de novo* long-read genome assembly, as it will enable significantly faster assembly of large genomes, but also the way we design, implement, and run genomics computation in general. This dissertation demonstrates how parallel computing can be beneficial for genomics and how it is possible to implement high-performance software while prioritizing programmer productivity and algorithm flexibility through appropriate abstraction. It also demonstrates how computing is moving toward massively parallel machines, where high-performance and parallel computing competencies and techniques will become increasingly important.

Bibliography

[1] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. "The transitive reduction of a directed graph". In: *SIAM Journal on Computing* 1.2 (1972), pp. 131–137.

[2] Sadaf R Alam et al. "Characterization of scientific workloads on systems with multi-core processors". In: *IEEE international symposium on workload characterization*. IEEE. 2006, pp. 225–236.

[3] Francis Alexander et al. "Exascale Applications: Skin in the Game". In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190056.

[4] George Almasi. *PGAS (Partitioned Global Address Space) Languages*. 2011.

[5] Dana Angluin and Leslie G Valiant. "Fast probabilistic algorithms for Hamiltonian circuits and matchings". In: *Journal of Computer and system Sciences* 18.2 (1979), pp. 155–193.

[6] Krste Asanovic et al. "The landscape of parallel computing research: A view from Berkeley". In: (2006).

[7] Baruch Awerbuch and Yossi Shiloach. "New connectivity and MSF algorithms for shuffle-exchange network and PRAM". In: *IEEE Transactions on Computers* 36.10 (1987), pp. 1258–1263.

[8] Ariful Azad and Aydın Buluç. "LACC: A Linear-Algebraic Algorithm for Finding Connected Components in Distributed Memory". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. ACM New York, NY, USA, 2019.

[9] Ariful Azad et al. "Combinatorial BLAS 2.0: Scaling Combinatorial Algorithms on Distributed-Memory Systems". In: *IEEE Transactions on Parallel & Distributed Systems* 33.4 (2022), pp. 989–1001. DOI: 10.1109/TPDS.2021.3094091.

[10] Ariful Azad et al. "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication". In: *SIAM Journal on Scientific Computing* 38.6 (2016), pp. C624–C651.

[11] Vlastimil Babka, Lukáš Marek, and Petr Tuma. "When misses differ: Investigating impact of cache misses on observed performance". In: *15th International Conference on Parallel and Distributed Systems*. IEEE. 2009, pp. 112–119.

[12] John Bachan et al. "UPC++: A high-performance communication framework for asynchronous computation". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 963–973.

[13] Grey Ballard et al. "Communication optimal parallel multiplication of sparse random matrices". In: *25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2013, pp. 222–231.

[14] Rodrigo P Baptista et al. "Assembly of highly repetitive genomes using short reads: the genome of discrete typing unit III Trypanosoma cruzi strain 231". In: *Microbial genomics* 4.4 (2018).

[15] Taylor Barnes et al. "Evaluating and optimizing the nersc workload on knights landing". In: *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE. 2016, pp. 43–53.

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

[17] Maciej Besta et al. "Communication-efficient jaccard similarity for high-performance distributed genome comparisons". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2020, pp. 1122–1132.

[18] Paola Bonizzoni et al. "An external-memory algorithm for string graph construction". In: *Algorithmica* 78.2 (2017), pp. 394–424.

[19] Paola Bonizzoni et al. "FSG: fast string graph construction for de novo assembly". In: *Journal of computational biology* 24.10 (2017), pp. 953–968.

[20] Benjamin Brock. "RDMA-Based Distributed Data Structures for Large-Scale Parallel Systems". In: (2022).

[21] Benjamin Brock, Aydın Buluç, and Katherine Yelick. "BCL: A cross-platform distributed data structures library". In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10.

[22] Benjamin Brock et al. "Considerations for a distributed GraphBLAS API". In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2020, pp. 215–218.

[23] Andrei Z Broder. "On the resemblance and containment of documents". In: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE. 1997, pp. 21–29.

[24] Aydin Buluc and John R Gilbert. "Challenges and advances in parallel sparse matrix-matrix multiplication". In: *37th International Conference on Parallel Processing*. IEEE. 2008, pp. 503–510.

[25] Aydin Buluc and John R Gilbert. "On the representation and multiplication of hypersparse matrices". In: *International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE. 2008.

[26] Aydin Buluç and John R Gilbert. "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments". In: *SIAM Journal on Scientific Computing* 34.4 (2012), pp. C170–C191.

[27] Aydin Buluç et al. "Design of the GraphBLAS API for C". In: *2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE. 2017, pp. 643–652.

[28] Aydın Buluç and John Gilbert. "New ideas in sparse matrix matrix multiplication". In: *Graph Algorithms in the language of linear algebra*. SIAM, 2011, pp. 315–337.

[29] Aydın Buluç and John R Gilbert. "The Combinatorial BLAS: Design, implementation, and applications". In: *The International Journal of High Performance Computing Applications* 25.4 (2011), pp. 496–509.

[30] Aydın Buluç and Kamesh Madduri. "Parallel breadth-first search on distributed memory systems". In: *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2011, pp. 1–12.

[31] Michael Burrows and David J Wheeler. *A block-sorting lossless data compression algorithm*. Tech. rep. Digital Equipment Corporation, 1994.

[32] Rajkumar Buyya et al. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility". In: *Future Generation computer systems* 25.6 (2009), pp. 599–616.

[33] William W Carlson et al. *Introduction to UPC and language specification*. Tech. rep. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[34] Antonio Bernardo Carvalho, Eduardo G Dupim, and Gabriel Goldstein. "Improved assembly of noisy long reads by k-mer validation". In: *Genome research* 26.12 (2016), pp. 1710–1720.

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

[36] Barbara Chapman et al. "Introducing OpenSHMEM: SHMEM for the PGAS community". In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. 2010, pp. 1–3.

[37] Wei-Yu Chen et al. "A performance analysis of the Berkeley UPC compiler". In: *Proceedings of the 17th annual international conference on Supercomputing*. 2003, pp. 63–73.

[38] Haoyu Cheng et al. "Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm". In: *Nature methods* 18.2 (2021), pp. 170–175.

[39] Herman Chernoff et al. "A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations". In: *The Annals of Mathematical Statistics* 23.4 (1952), pp. 493–507.

[40] Chen-Shan Chin et al. "Phased diploid genome assembly with single-molecule real-time sequencing". In: *Nature methods* 13.12 (2016), pp. 1050–1054.

[41] Justin Chu et al. "Innovations and challenges in detecting long read overlaps: an evaluation of the state-of-the-art". In: *Bioinformatics* 33.8 (2016), pp. 1261–1270.

[42] Phillip Colella. *Defining software requirements for scientific computing*. 2004.

[43] Phillip EC Compeau, Pavel A Pevzner, and Glenn Tesler. "Why are de Bruijn graphs useful for genome assembly?" In: *Nature biotechnology* 29.11 (2011), p. 987.

[44] Cray. *Cray XC Series Network*. `https://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf`.

[45] Mark Crovella et al. "Using communication-to-computation ratio in parallel program design and performance prediction". In: *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*. IEEE. 1992, pp. 238–245.

[46] David E Culler et al. "Parallel programming in Split-C". In: *Supercomputing'93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. IEEE. 1993, pp. 262–273.

[47] Timothy A Davis. "Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra". In: *ACM Transactions on Mathematical Software (TOMS)* 45.4 (2019), pp. 1–25.

[48] Timothy A Davis. *Direct methods for sparse linear systems*. SIAM, 2006.

[49] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. "Performance-portable sparse matrix-matrix multiplication for many-core architectures". In: *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2017, pp. 693–702.

[50] Douglas Doerfler et al. "Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor". In: *High Performance Computing*. Ed. by Michela Taufer, Bernd Mohr, and Julian M. Kunkel. Cham: Springer International Publishing, 2016, pp. 339–353. ISBN: 978-3-319-46079-6.

[51] Douglas Doerfler et al. "Evaluating the networking characteristics of the Cray XC-40 Intel Knights Landing-based Cori supercomputer at NERSC". In: *Concurrency and Computation: Practice and Experience* 30.1 (2018), e4297.

[52] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. "The LINPACK benchmark: past, present and future". In: *Concurrency and Computation: practice and experience* 15.9 (2003), pp. 803–820.

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

[54] Robert Edgar. "Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences". In: *PeerJ* 9 (2021), e10805.

[55] Jack Edmonds and Ellis L Johnson. "Matching: A well-solved class of integer linear programs". In: *Combinatorial Optimization—Eureka, You Shrink!* Springer, 2003, pp. 27–30.

[56] John Eid et al. "Real-time DNA sequencing from single polymerase molecules". In: *Science* 323.5910 (2009), pp. 133–138.

[57] Marquita Ellis et al. "diBELLA: Distributed Long Read to Long Read Alignment". In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–11.

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

[59] Marquita May Ellis. *Parallelizing irregular applications for distributed memory scalability: Case studies from genomics*. University of California, Berkeley, 2020.

[60] Constantinos Evangelinos and Chris Hill. "Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2". In: *CCA-08* 2.2.40 (2008), pp. 2–34.

[61] *FASTA format*. https://zhanggroup.org/FASTA/. (Accessed on 07/26/2022).

[62] Paolo Ferragina and Giovanni Manzini. "Indexing compressed text". In: *Journal of the ACM (JACM)* 52.4 (2005), pp. 552–581.

[63] Michael J Flynn. "Some computer organizations and their effectiveness". In: *IEEE transactions on computers* 100.9 (1972), pp. 948–960.

[64] Armando Fox et al. "Above the clouds: A Berkeley view of cloud computing". In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28.13 (2009).

[65] Matteo Frigo. "A fast Fourier transform compiler". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1999, pp. 169–180.

[66] Matteo Frigo and Steven G Johnson. "FFTW: An adaptive software architecture for the FFT". In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*. Vol. 3. IEEE. 1998, pp. 1381–1384.

[67] Matteo Frigo and Steven G Johnson. *The fastest fourier transform in the west*. Tech. rep. Massachusetts Institute of Technology, Cambridge, 1997.

[68] GASNet. *GASNet-EX Performance Examples*. https://gasnet.lbl.gov/performance/.

[69] Evangelos Georganas et al. "Extreme scale de novo metagenome assembly". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. IEEE Computer Society, 2018, pp. 122–134.

[70] Evangelos Georganas et al. "HipMer: an extreme-scale de novo genome assembler". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2015, p. 14.

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

[72] Tarek El-Ghazawi et al. *UPC: distributed shared memory programming*. John Wiley & Sons, 2005.

[73] Priyanka Ghosh, Sriram Krishnamoorthy, and Ananth Kalyanaraman. "PaKman: A Scalable Algorithm for Generating Genomic Contigs on Distributed Memory Machines". In: *IEEE Transactions on Parallel and Distributed Systems* 32.5 (2020), pp. 1191–1209.

[74] John R Gilbert, Cleve Moler, and Robert Schreiber. "Sparse matrices in MATLAB: Design and implementation". In: *SIAM journal on matrix analysis and applications* 13.1 (1992), pp. 333–356.

[75] Francesca Giordano et al. "De novo yeast genome assemblies from MinION, PacBio and MiSeq platforms". In: *Scientific reports* 7.1 (2017), p. 3935.

[76] Joseph E Gonzalez et al. "Graphx: Graph processing in a distributed dataflow framework". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014, pp. 599–613.

[77] Sara Goodwin et al. "Oxford Nanopore sequencing, hybrid error correction, and de novo assembly of a eukaryotic genome". In: *Genome research* 25.11 (2015), pp. 1750–1756.

[78] John E Gorzynski et al. "Ultrarapid Nanopore Genome Sequencing in a Critical Care Setting". In: *The New England journal of medicine* (2022).

[79] Ronald L Graham. "Bounds for certain multiprocessing anomalies". In: *Bell system technical journal* 45.9 (1966), pp. 1563–1581.

[80] Ronald L. Graham. "Bounds on multiprocessing timing anomalies". In: *SIAM journal on Applied Mathematics* 17.2 (1969), pp. 416–429.

[81] Ananth Grama et al. *Introduction to parallel computing*. Pearson Education, 2003.

[82] William Gropp et al. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999.

[83] Giulia Guidi et al. "BELLA: Berkeley Efficient Long-Read to Long-Read Aligner and Overlapper". In: *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)* (2021), pp. 123–134.

[84] Giulia Guidi et al. "Distributed-Memory Parallel Contig Generation for *De Novo* Long-Read Genome Assembly". In: *Proceedings of the 51th International Conference on Parallel Processing (ICPP)*. 2022.

[85] Giulia Guidi et al. "Parallel String Graph Construction and Transitive Reduction for De Novo Genome Assembly". In: *International Parallel and Distributed Processing Symposium (IPDPS)* (2021), pp. 517–526.

[86] Abhishek Gupta et al. "Evaluating and improving the performance and scheduling of HPC applications in cloud". In: *IEEE Transactions on Cloud Computing* 4.3 (2014), pp. 307–321.

[87] Alexey Gurevich et al. "QUAST: quality assessment tool for genome assemblies". In: *Bioinformatics* 29.8 (2013), pp. 1072–1075.

[88] Fred G Gustavson. "Two fast algorithms for sparse matrices: Multiplication and permuted transposition". In: *ACM Transactions on Mathematical Software (TOMS)* 4.3 (1978), pp. 250–269.

[89] Kim Hazelwood et al. "Applied machine learning at facebook: A datacenter infrastructure perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 620–629.

[90] Qiming He et al. "Case study for running HPC applications in public clouds". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. 2010, pp. 395–401.

[91] National Institute of Health. *The Cost of Sequencing a Human Genome*. `https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost`. (Accessed on 06/15/2022).

[92] Wassily Hoeffding. "Probability inequalities for sums of bounded random variables". In: *Journal of the American statistical association* 58.301 (1963), pp. 13–30.

[93] Steven Hofmeyr et al. "Terabase-scale metagenome coassembly with MetaHipMer". In: *Scientific reports* 10.1 (2020), pp. 1–11.

[94] Shigeru Ida and Junichiro Makino. "N-Body simulation of gravitational interaction between planetesimals and a protoplanet: I. velocity distribution of planetesimals". In: *Icarus* 96.1 (1992), pp. 107–120.

[95] Benjamin G Jackson and Srinivas Aluru. "Parallel construction of bidirected string graphs for genome assembly". In: *37th International Conference on Parallel Processing*. IEEE. 2008, pp. 346–353.

[96] Miten Jain et al. "The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community". In: *Genome Biology* 17.1 (2016), p. 239.

[97] Norman P Jouppi et al. "A domain-specific supercomputer for training deep neural networks". In: *Communications of the ACM* 63.7 (2020), pp. 67–78.

[98] Jeremy Kepner et al. "Mathematical foundations of the GraphBLAS". In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2016, pp. 1–9.

[99] Mikhail Kolmogorov et al. "Assembly of long, error-prone reads using repeat graphs". In: *Nature biotechnology* 37.5 (2019), pp. 540–546.

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

[101] Monica D Lam, Edward E Rothberg, and Michael E Wolf. "The cache performance and optimizations of blocked algorithms". In: *ACM SIGOPS Operating Systems Review* 25.Special Issue (1991), pp. 63–74.

[102] Chuck L Lawson et al. "Basic linear algebra subprograms for Fortran usage". In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323.

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

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

[105] Zhenyu Li et al. "Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph". In: *Briefings in functional genomics* 11.1 (2012), pp. 25–37.

[106] *libc/bionic/pthread.c - platform/bionic - Git at Google*. https://android.googlesource.com/platform/bionic/+/10ce969/libc/bionic/pthread.c. (Accessed on 08/09/2022).

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

[108] David J Lipman and William R Pearson. "Rapid and sensitive protein similarity searches". In: *Science* 227.4693 (1985), pp. 1435–1441.

[109] Nicholas J Loman, Joshua Quick, and Jared T Simpson. "A complete bacterial genome assembled de novo using only nanopore sequencing data". In: *Nature methods* 12.8 (2015), pp. 733–735.

[110] Andrey Markov. "Extension of the limit theorems of probability theory to a sum of variables connected in a chain". In: (1971).

[111] John D McCalpin. "STREAM benchmark". In: 22 (1995). URL: http://www.cs.virginia.edu/stream/ref.html.

[112] Paul Medvedev et al. "Computability of models for sequence assembly". In: *International Workshop on Algorithms in Bioinformatics*. Springer. 2007, pp. 289–301.

[113] Peter Mell and Tim Grance. "The NIST definition of cloud computing". In: (2011).

[114] Pall 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.

[115] Jason R Miller et al. "Aggressive assembly of pyrosequencing reads with mates". In: *Bioinformatics* 24.24 (2008), pp. 2818–2824.

[116] Philip J. Mucci. *LLCbench Home Page*. http://icl.cs.utk.edu/llcbench/index.htm.

[117] Hamid Mushtaq et al. "Sparkga: A spark framework for cost effective, fast and accurate dna analysis at scale". In: *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*. 2017, pp. 148–157.

[118] Eugene W Myers. "An O(ND) difference algorithm and its variations". In: *Algorithmica* 1.1-4 (1986), pp. 251–266.

[119] Eugene W Myers. "The fragment assembly string graph". In: *Bioinformatics* 21.suppl_2 (2005), pp. ii79–ii85.

[120] Gene Myers. "Efficient local alignment discovery amongst noisy long reads". In: *International Workshop on Algorithms in Bioinformatics*. Springer. 2014, pp. 52–67.

[121] Niranjan Nagarajan and Mihai Pop. "Parametric complexity of sequence assembly: theory and applications to next generation sequencing". In: *Journal of computational biology* 16.7 (2009), pp. 897–908.

[122] Yusuke Nagasaka et al. "Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors". In: *Parallel Computing* (2019), p. 102545.

[123] NERSC.

[124] NERSC. *Interconnect - NERSC Documentation*. https://docs.nersc.gov/systems/cori/interconnect/.

[125] Marco AS Netto et al. "HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges". In: *ACM Computing Surveys (CSUR)* 51.1 (2018), pp. 1–29.

[126] *NIH Researchers Deploy Nanopore Sequencing for Large-Scale Alzheimer's Study, Genomeweb*. https://www.genomeweb.com/sequencing/nih-researchers-deploy-nanopore-sequencing-large-scale-alzheimers-study. (Accessed on 07/27/2022).

[127] Sergey Nurk et al. "HiCanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads". In: *Genome research* 30.9 (2020), pp. 1291–1305.

[128] Yukiteru Ono, Kiyoshi Asai, and Michiaki Hamada. "PBSIM: PacBio reads simulator—toward accurate genome assembly". In: *Bioinformatics* 29.1 (2012), pp. 119–121.

[129] *OpenMP.org » About the OpenMP ARB and OpenMP.org*. https://web.archive.org/web/20130809153922/http://openmp.org/wp/about-openmp/. (Accessed on 08/09/2022).

[130] *PacificBiosciences/pb-assembly: PacBio Assembly Tool Suite*. https://github.com/PacificBiosciences/pb-assembly. (Accessed on 07/25/2022).

[131] David Padua. *Encyclopedia of parallel computing*. Springer Science & Business Media, 2011.

[132] Dhabaleswar K. Panda. *OSU Micro-Benchmarks*. 2018.

[133] Alexander J Paul et al. "SORA: Scalable Overlap-Graph Reduction Algorithms for Genome Assembly using Apache Spark in the Cloud". In: *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (2018), pp. 718–723.

[134] Adam M Phillippy, Michael C Schatz, and Mihai Pop. "Genome assembly forensics: finding the elusive mis-assembly". In: *Genome biology* 9.3 (2008), R55.

[135] Jean-Eric Pin. *Tropical semirings*. 1998.

[136] Mohammad Pourkheirandish et al. "Global role of crop genomics in the face of climate change". In: *Frontiers in Plant Science* (2020), p. 922.

[137] Daniel Reed, Dennis Gannon, and Jack Dongarra. "Reinventing High Performance Computing: Challenges and Opportunities". In: *arXiv preprint arXiv:2203.02544* (2022).

[138] David C Rees, Thomas N Williams, and Mark T Gladwin. "Sickle-cell disease". In: *The Lancet* 376.9757 (2010), pp. 2018–2031.

[139] James Reinders. "VTune performance analyzer essentials". In: *Intel Press* (2005).

[140] Anthony Rhoads and Kin Fai Au. "PacBio sequencing and its applications". In: *Genomics, Proteomics & Bioinformatics* 13.5 (2015), pp. 278–289.

[141] Michael Roberts et al. "Reducing storage requirements for biological sequence comparison". In: *Bioinformatics* 20.18 (2004), pp. 3363–3369.

[142] Jue Ruan and Heng Li. "Fast and accurate long-read assembly with wtdbg2". In: *Nature methods* 17.2 (2020), pp. 155–158.

[143] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[144] Oguz Selvitopi et al. "Distributed Many-to-Many Protein Sequence Alignment Using Sparse Matrices". In: *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2020).

[145] Kishwar Shafin et al. "Nanopore sequencing and the Shasta toolkit enable efficient de novo assembly of eleven human genomes". In: *Nature biotechnology* 38.9 (2020), pp. 1044–1053.

[146] Timothy Sherwood, Brad Calder, and Joel Emer. "Reducing cache misses using hardware and software page placement". In: *Proceedings of the 13th international conference on Supercomputing*. 1999, pp. 155–164.

[147] Anton Shterenlikht. "Fortran coarray library for 3D cellular automata microstructure simulation". In: *7th International Conference on PGAS Programming Models*. 2014, p. 16.

[148] Jared T Simpson and Richard Durbin. "Efficient de novo assembly of large genomes using compressed data structures". In: *Genome research* 22.3 (2012), pp. 549–556.

[149] Jouni Sirén et al. "Pangenomics enables genotyping of known structural variants in 5202 diverse genomes". In: *Science* 374.6574 (2021), abg8871.

[150] Ivan Sović et al. "Fast and sensitive mapping of nanopore sequencing reads with GraphMap". In: *Nature communications* 7.1 (2016), pp. 1–11.

[151] *The Linpack Benchmark — TOP500*. https://top500.org/project/linpack/. (Accessed on 07/26/2022).

[152] Marta Tomaszkiewicz et al. "A time-and cost-effective strategy to sequence mammalian Y Chromosomes: an application to the de novo assembly of gorilla Y". In: *Genome research* 26.4 (2016), pp. 530–540.

[153] Top500. *Cori - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect — TOP500*. https://www.top500.org/system/178924/.

[154] George Tsouloupas and Marios D Dikaiakos. "Characterization of computational grid resources using low-level benchmarks". In: *Second IEEE International Conference on e-Science and Grid Computing*. IEEE. 2006, pp. 70–70.

[155] Robert A Van De Geijn and Jerrell Watts. "SUMMA: Scalable universal matrix multiplication algorithm". In: *Concurrency: Practice and Experience* 9.4 (1997), pp. 255–274.

[156] Robert Vaser and Mile Šikić. "Raven: a de novo genome assembler for long reads". In: *BioRxiv* (2021), pp. 2020–08.

[157] Vincent M Weaver. "Linux perf_event features and overhead". In: *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*. Vol. 13. 2013.

[158] Aaron M Wenger et al. "Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome". In: *Nature biotechnology* 37.10 (2019), pp. 1155–1162.

[159] Steven Cameron Woo et al. "The SPLASH-2 programs: Characterization and methodological considerations". In: *ACM SIGARCH computer architecture news* 23.2 (1995), pp. 24–36.

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

[161] Katherine Yelick et al. "The Magellan Report on Cloud Computing for Science". In: vol. 3. 2011.

[162] Katherine Yelick et al. "The Parallelism Motifs of Genomic Data Analysis". In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190394.

[163] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple Linux utility for resource management". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60.

[164] Matei Zaharia et al. "Apache spark: a unified engine for big data processing". In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

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