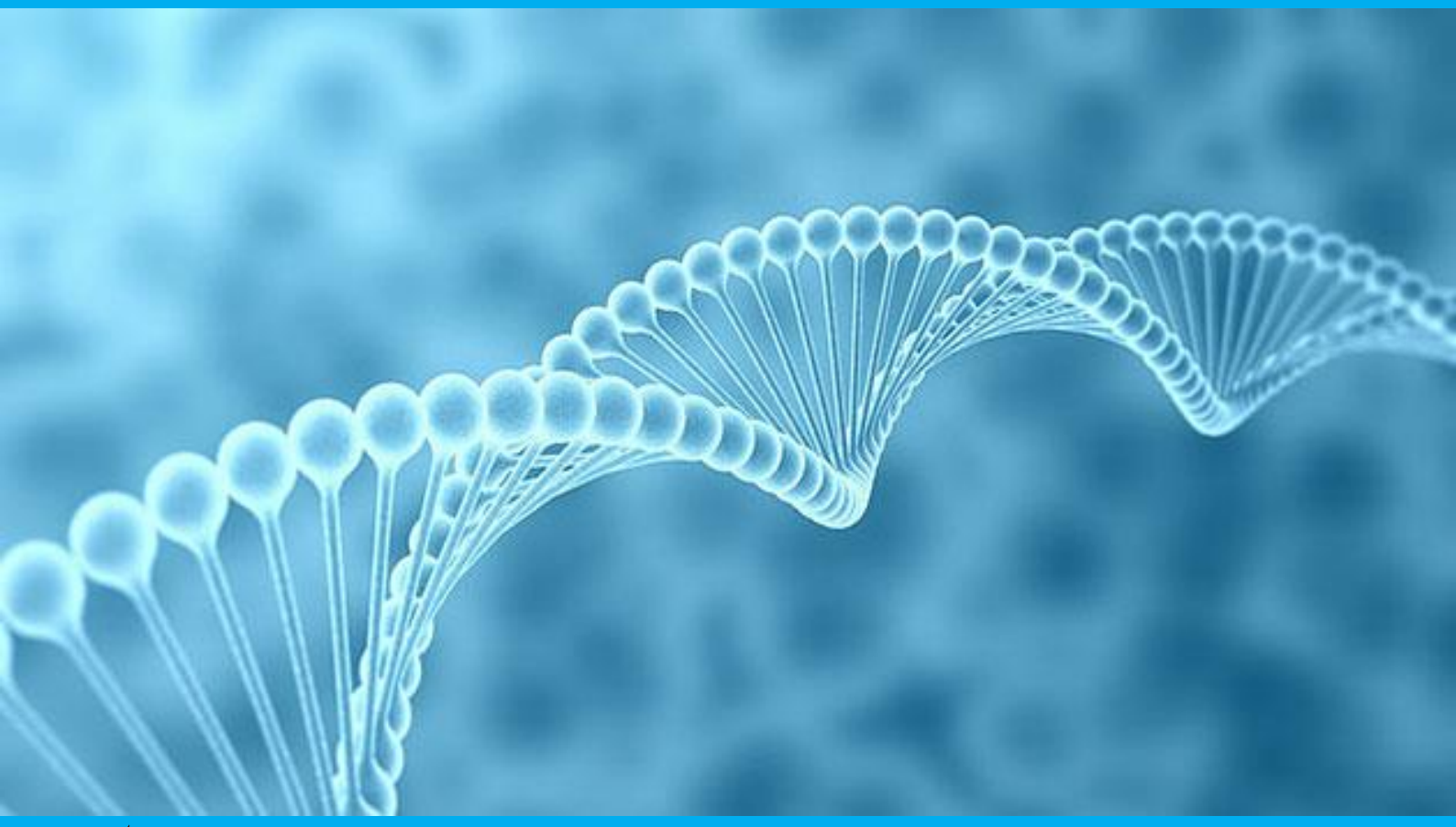
High-Performance Optimization of DNA Long Read De Novo Assembler

Kaiyi Zhao



TU Delft

High-Performance Optimization of DNA Long Read De Novo Assembler

by

Kaiyi Zhao

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday August 29, 2024 at 13:30 PM.

Student number: 5797594
Project duration: November 6, 2023 – August 29, 2024
Thesis committee: Dr. Zaid Al-Ars, Computer Engineering Lab, TU Delft, supervisor
Dr. Jasmijn Baaijens, Bioinformatics Lab, TU Delft
Dr. Tanveer Ahmad, National Cancer Institute, National Institutes of Health

This thesis is confidential and cannot be made public until August 28, 2024.

TU Delft

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

This thesis focuses on accelerating the polishing stage of the Flye genome assemblers. Flye is a de novo assembler designed for long reads produced by modern sequencing technologies, excelling in handling large genomes with high accuracy and efficiency. A crucial component of the assembly process is the polishing stage, which refines the draft assembly to correct errors and improve overall accuracy. However, this stage is computationally intensive and time-consuming, presenting a significant bottleneck in genome assembly workflows.

To address this, a novel multi-threading architecture is introduced, significantly reducing mutex contention by minimizing the use and acquisition times of mutexes within the bubble processor. Additionally, advanced vectorization techniques using AVX (Advanced Vector Extensions) instructions are incorporated to process multiple reads simultaneously. These optimizations effectively parallelize the polishing process and exploit modern CPU capabilities for enhanced performance.

Benchmarking the enhanced polishing stage on both bacteria and human genome datasets demonstrates a substantial improvement in processing time. For the bacteria dataset, the error correction process achieves speedups of 3.0x and 4.3x using AVX2 and AVX-512 instructions running on one core, respectively. The process realizes speedups of 2.6x and 2.7x with AVX2 and AVX-512 running on eight cores. For the human genome dataset, the process demonstrates a speedup of 4.0x when handling 1 million bubbles running on one core, while 32 cores yield a speedup of 2.3x for the same dataset. Applying AVX2 to the complete dataset on 64 cores results in a speedup of 1.4x. This acceleration not only reduces computational costs but also expedites the overall genome assembly process, making it more feasible for large-scale and time-sensitive genomic studies. The implementation is available on GitHub.

Acknowledgments

I would like to express my deepest gratitude to those who have supported and guided me throughout my master's thesis journey.

Dr. Zaid Al-Ars, my thesis advisor, has been an unwavering source of guidance and support. Our weekly meetings were instrumental in keeping my research on track, and his insightful advice helped me navigate through various challenges. His dedication and expertise have been invaluable from the start to the completion of my thesis.

Dr. Tanveer Ahmad, my external supervisor, played a crucial role in helping me quickly immerse myself in this research area. His guidance throughout the research process was essential in shaping the direction and outcome of my work. I am grateful for his mentorship and the time he devoted to ensuring my success.

Dr. Jasmijn Baaijens, a member of my thesis committee, took time out of her busy schedule to be part of this important milestone in my academic journey. I sincerely appreciate her commitment and her valuable contributions to my thesis.

To my family, who have been a constant source of strength and encouragement, I owe a profound debt of gratitude. During my master's studies, I experienced the loss of my grandmother and grandfather. Their love and support have always been with me, and I know they would be proud of this achievement. My family's persistent support helped me complete this thesis.

Lastly, I want to thank my roommate, Tian, who has been more than just a friend. We first met during our bachelor's studies, and together we embarked on this journey to the Netherlands to pursue our master's degrees. Over the past two years, we have studied together, played volleyball, and even traveled on vacations. We have supported each other through the ups and downs of our thesis work, and I am grateful for his companionship. I am thrilled that we can celebrate our graduation together.

1 Introduction

1.1. Context

Genome assembly is a vital process in genomics that reconstructs an organism's complete genome sequence from short DNA fragments, similar to solving a complex puzzle. Advances in high-throughput sequencing technologies have dramatically increased the amount of data available, generating millions of short sequences, or "reads." These require advanced computational tools to accurately assemble the genome. There are two main types of genome assembly: de novo assembly, which builds genomes without a reference, and reference-guided assembly, which uses a related genome as a guide to enhance accuracy and reduce complexity. Genome assembly is crucial beyond basic research, impacting evolutionary biology, medicine, and agriculture by facilitating the identification of genetic variations, understanding evolutionary relationships, and improving crops through genetic engineering.

Accurate genome assembly faces challenges due to repetitive regions. While long single-molecule sequencing reads can resolve genomic repeats more effectively than short-read data, many long-read assembly algorithms lack the necessary repeat characterization for optimal assemblies. Flye [10] is a long-read de novo assembly algorithm that generates arbitrary paths in an unknown repeat graph, termed disjointigs, and then constructs a precise repeat graph from these error-prone disjointigs. Flye is evaluated against five leading assemblers, demonstrating that it produces superior or comparable assemblies with significantly greater speed.

Polishing is one of the most time-consuming components in the Flye workflow. This crucial step in genome assembly greatly improves the accuracy of the draft genome by correcting sequencing errors. ABruijn, as first introduced by Lin et al. [14], included a polisher. Flye's polisher builds upon this foundation, enhancing both accuracy and speed. ABruijn employs the BLASR algorithm [3], while Flye utilizes minimap2 [13] to align sequencing reads against the draft genome. Initially, these alignments are prone to inaccuracies due to the error-prone nature of the draft genome. To address this, the alignments are refined to achieve precise error correction. This method involves partitioning the multiple alignments of reads into shorter segments, known as mini-alignments. Each mini-alignment segment is then individually error-corrected to produce a consensus sequence.

1.2. Challenges, Problem Statement and Research Questions

Although the Flye polisher leverages advanced heuristics and sophisticated software libraries, its performance significantly lags behind the optimal computing capabilities of modern CPUs. Significant research efforts have been dedicated to accelerating various stages of reference-based short-read sequencing, such as the Burrows-Wheeler Aligner (BWA) [2] [8], as well as haplotype calling [1] [18]. However, the acceleration of de novo-based long-read assembly remains an ongoing area of research. Currently, there is limited research aimed at enhancing Flye's computational efficiency. One notable attempt focuses on improving its memory usage [7]. However, efforts to enhance CPU utilization and reduce the overall computation time remain largely unexplored.

Genome assembly is a foundational process in genomic research, allowing researchers to reconstruct complete genomes from sequence data. The polishing phase, which corrects errors in the initial assembly, is crucial for ensuring the accuracy and completeness of the final genome. However, this

phase is often computationally intensive and time-consuming, posing a significant bottleneck in genome assembly pipelines. Flye is a leading long-read de novo assembler known for its ability to construct repeat graphs and assemble complex genomes effectively. Despite its strengths, Flye faces significant performance challenges during the polishing phase. These challenges primarily stem from suboptimal utilization of modern CPU capabilities, resulting in inefficiencies that hinder its overall performance. Addressing these inefficiencies is essential for improving the speed and accuracy of genome assemblies. This thesis aims to identify and address specific limitations in Flye's polishing phase to optimize CPU resource usage and enhance the overall performance of the genome assembly process.

To guide this investigation, we propose the following research questions:

1. **How can a new multi-threading architecture be designed to minimize mutex contention and enhance parallel processing in the Flye polishing step?** This question focuses on improving thread management and synchronization to utilize CPU cores more efficiently.

2. **What is the impact of using modern SIMD instruction sets (e.g., AVX2 and AVX-512) on the performance of the Flye polisher?** This question explores how advanced vector processing units can accelerate error correction by processing multiple data points simultaneously.

3. **How does the performance of the optimized polisher compare to the original Flye polisher in terms of processing speed and accuracy?** This question aims to quantify the improvements in speed and verify that optimizations maintain or improve assembly accuracy.

4. **What are the trade-offs between computational speed and resource usage in the optimized Flye polisher?** This question investigates the balance between increased processing speed and demands on computational resources, such as memory and CPU load.

By addressing these questions, this thesis seeks to significantly improve the efficiency of genome polishing processes, thereby enhancing the overall performance of genome assembly pipelines and facilitating more rapid and accurate genomic research.

1.3. Contribution

This thesis aims to enhance the error correction process of the polisher. To expedite the error correction step, we developed a parallel polishing algorithm adopting a new multi-threading architecture and utilizing single-instruction multiple-data (SIMD) techniques, leveraging modern CPUs' vector processing units (VPUs).

A multithreading architecture is designed to create and manage jobs across multiple cores, enabling the utilization of all available cores on a CPU which significantly enhances performance by parallelizing tasks. In original multithreaded architecture, a mutex (mutual exclusion) is used to prevent concurrent access to shared resources, ensuring data integrity. However, using mutexes can introduce performance bottlenecks due to increased waiting times when multiple threads attempt to access the same resource. Our new architecture minimizes the critical sections controlled by mutexes and reduces the number of times these mutexes need to be accessed. This improvement is particularly beneficial when handling large number of threads, as it decreases contention and overhead, resulting in better overall performance.

SIMD instructions are designed to simultaneously perform the same operation across multiple data points, thereby enabling parallel computation. Although Streaming SIMD Extensions (SSE) offer a 128-bit SIMD instruction set, they were not suitable for our needs, as our processing requires handling 64-bit data. With SSE, we could only process two data items at a time, which did not yield significant performance improvements due to the associated overhead. Modern CPUs feature Advanced Vector Extensions (AVX2 and AVX-512), which support 256-bit and 512-bit SIMD instructions, respectively. These extensions allow for the simultaneous processing of four and eight 64-bit data items, providing substantial performance gains despite the overhead involved in using SIMD instructions.

In all of our optimizations, we ensured that the final output remained entirely consistent with that of Flye, enabling users to effortlessly switch to a faster version of Flye whenever increased computational speed is needed. We evaluated our optimized polisher implementation against the original by testing it on Human002 Oxford Nanopore (Ultra-long GridION data) [23]. Our results showed up to a 1.4-fold increase in processing speed.

1.4. Thesis Layout

This thesis is organized into six chapters, each focusing on different aspects of the research to guide the reader through the process of optimizing the Flye polishing phase.

Chapter 1 sets the context for our research by discussing the importance of DNA assembly in genomics and the role of the Flye assembly algorithm, with a focus on its polishing phase. We highlight the challenges associated with this process, particularly the time-consuming nature of polishing and the lack of existing work aimed at improving its performance. To address these challenges, we present four research questions that guide our investigation. The core contribution of this thesis is to enhance the efficiency of the polishing phase by improving multithreading capabilities and leveraging SIMD techniques, ensuring that the speedup is achieved without altering the accuracy of the output.

Chapter 2 covers the key components of DNA long-read sequencing and genome assembly. We begin by discussing the advantages and disadvantages of long-read sequencing. Next, we outline the assembly pipeline, from basecalling to polishing, and highlight the Flye assembly algorithm, focusing on its workflow and polishing capabilities. We also compare sequence alignment algorithms, namely Needleman-Wunsch and Smith-Waterman, and describe the application of the Needleman-Wunsch algorithm in Flye's polisher. Additionally, we explore parallel computing techniques like multi-threading and SIMD, which enhance computational efficiency, and introduce Amdahl's Law to explain the potential gains from parallelization in genome assembly processes.

Chapter 3 details the implementation and optimization of various components to enhance the polishing process. We begin by discussing the General Polisher, outlining its purpose, initial performance enhancements, and the application of SIMD optimization to speed up processing. Next, we introduce the Dinucleotide Fixer, explaining its purpose and the implementation of a separate alignment class tailored for this function. Finally, we describe the Bubble Processor, focusing on its role in resolving complex assembly issues and the development of new multi-threaded architectures to improve efficiency.

Chapter 4 describes the experimental setup used to evaluate the performance of our proposed optimizations in the Flye assembler. We profile the Flye algorithm using both bacterial and human genome datasets, analyzing the time distribution across different stages of the assembler with detailed attention to the polishing steps. Performance comparisons are made between different designs, emphasizing runtime and CPU utilization with varying numbers of threads and different SIMD instructions (AVX2, AVX-512).

Chapter 5 presents and interprets the results of the experiments, comparing the performance of the original and optimized Flye polishers. It explores the implications of these findings for genome assembly efficiency and accuracy.

Chapter 6 summarizes the key findings and contributions of the thesis. It discusses the broader impact on genomic research and suggests directions for future work, including potential technologies to optimize flye's polisher.

2

Background

2.1. DNA long-read sequencing

DNA long-read sequencing is an innovative technology that has transformed genomic research by providing much longer sequences of DNA compared to traditional short-read methods. This approach enables more accurate analysis of complex genomic regions, structural variations, and repetitive sequences that are difficult to resolve with short reads. The main platforms for long-read sequencing are Pacific Biosciences (PacBio) and Oxford Nanopore Technologies (ONT). PacBio uses Single Molecule Real-Time (SMRT) technology [5], producing reads over 10,000 base pairs long, and sometimes exceeding 100,000 base pairs. ONT employs nanopore sequencing [4], which measures electrical changes as DNA passes through a nanopore, offering reads that can exceed hundreds of kilobases.

Long-read sequencing provides several advantages over short-read methods. It improves genome assembly and the detection of structural variations such as insertions, deletions, inversions, and translocations. It also resolves highly repetitive regions, offering clearer insights into genome architecture, and facilitates haplotype phasing, which is important for understanding genetic diversity and disease associations [15]. Despite these advantages, long-read sequencing faces challenges, such as higher error rates compared to short-read sequencing, especially with ONT. However, ongoing improvements in sequencing chemistry, bioinformatics tools, and error correction algorithms are addressing these issues. The integration of long-read sequencing with complementary technologies, along with reductions in cost and improvements in read accuracy, is expected to expand its accessibility and application across various research domains [6].

2.2. Assembly Pipeline: From Basecalling to Polishing

As shown in Figure 2.1, the assembly pipeline for long-read sequencing transforms raw data into high-quality genome assemblies through a series of critical steps: basecalling, read correction, assembly, and assembly refinement.

Basecalling is the initial step, where raw signals from sequencing platforms, such as the electrical signals in Oxford Nanopore Technologies (ONT) or the fluorescence signals in Pacific Biosciences (PacBio), are converted into nucleotide sequences. Advanced machine learning algorithms, including neural networks, are employed to enhance the accuracy of these conversions, producing sequences accompanied by quality scores that guide subsequent steps.

Following basecalling, read correction addresses the inherent higher error rates of long-read sequencing. This process involves detecting and correcting common sequencing errors like insertions, deletions, and substitutions. Tools such as Canu [11] and FMLRC2 [16] align reads to each other to identify consensus sequences and correct errors, sometimes incorporating short-read data to leverage its high accuracy.

The corrected reads are then assembled into a coherent genome sequence. This involves identifying overlaps between reads and constructing contigs—long, contiguous sequences—using overlap information. Assemblers like Flye [10] and wtdbg2 [19] often utilize graph-based techniques to manage complex regions and ensure a robust assembly structure.

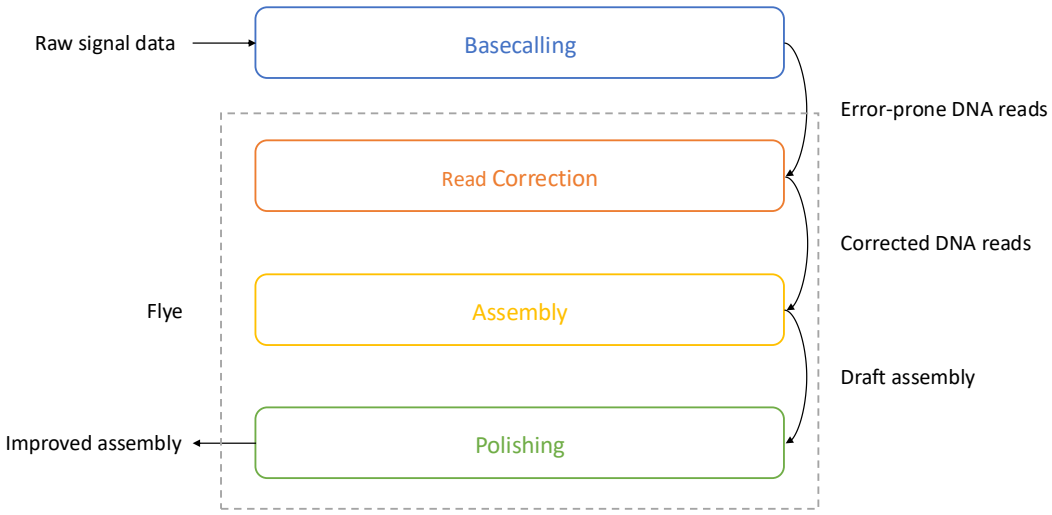


Figure 2.1: The different stages of the genome assembly pipeline

Finally, the polishing step refines the assembled genome, correcting remaining errors and improving the overall quality. Polishing tools, such as Racon [21] and Medaka align the original reads back to the contigs to detect and correct discrepancies, often using iterative processes to enhance accuracy. Additional data, like short reads, may be integrated during polishing to further improve the assembly. Through these steps, the assembly pipeline efficiently transforms raw sequencing data into high-quality genomic sequences, enabling detailed exploration of complex genomic structures.

2.3. Assembly algorithm: Flye

Workflow of Flye

Flye is an advanced assembly pipeline specifically designed for long-read sequencing data, offering an efficient and streamlined process for generating high-quality genome assemblies. The assembly process in long-read sequencing involves several critical stages, including basecalling, read correction, assembly, and assembly refinement. Flye is capable of handling raw reads directly from the basecalling stage, thereby simplifying the pipeline by removing the need for a separate read correction step. This makes Flye particularly advantageous for users seeking to minimize preprocessing steps without compromising on the quality of the assembly. Flye also integrates an intrinsic polishing stage, which refines the initial assembly by correcting errors and improving the overall accuracy of the output. This polishing step is crucial in enhancing the quality of the final contigs, ensuring they are polished and reliable for subsequent analysis. By combining these functionalities, Flye serves as a complete solution for long-read sequencing assembly, transforming raw sequence data into polished contigs with minimal user intervention.

There are five stages in flye workflow: Assembly, Consensus, Repeat, Contigger, and Polishing. It begins by identifying solid k-mers, which are k-mers with sufficient frequency to minimize errors and extends contigs by detecting overlaps between reads. The initial assembly stage may contain misassemblies, as repeats are not yet resolved. Flye then refines the assembly by aligning reads to the draft contigs using minimap2, improving accuracy through consensus sequence calling. In the repeat analysis phase, Flye constructs a repeat graph, collapsing and resolving repeats with the help of read information and graph structure. This process produces contiguous sequences representing genome segments. Finally, Flye polishes the assembly by aligning all reads to the current assembly and correcting errors using a maximum likelihood approach, with additional polishing cycles enhancing quality further. The first four stages in flye workflow is matched to the read correction and assembly

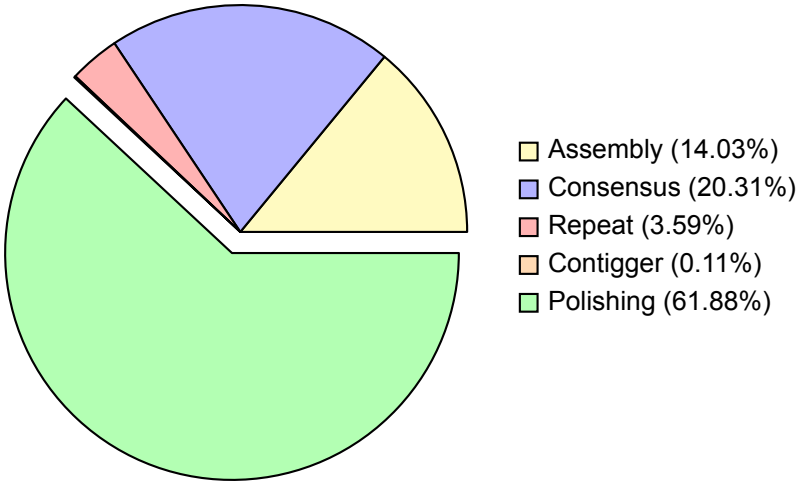


Figure 2.2: Time distribution across different stages of the Flye assembler on bacteria genome dataset

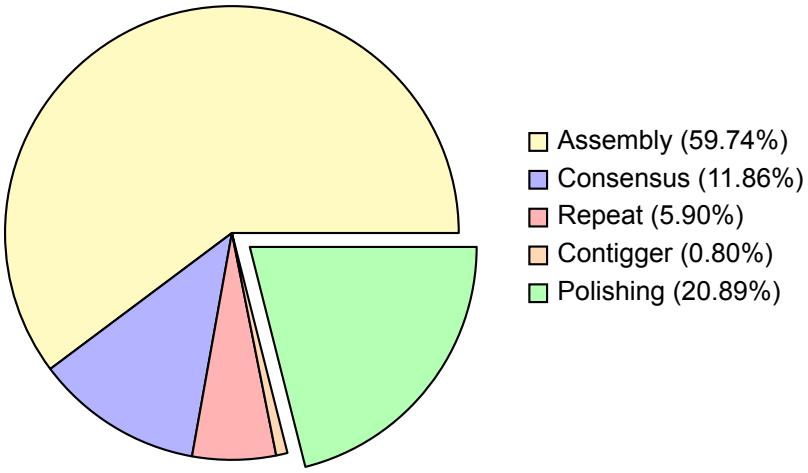


Figure 2.3: Time distribution across different stages of the Flye assembler on human genome dataset

steps in the assembly pipeline. And the last stage in flye workflow is matched to the polishing step in the assembly pipeline.

The time required for the assembly of different genomes varies significantly, and this variation is also reflected in the time distribution across all five stages of the process. For instance, as shown in Figure 2.2, the assembly of a bacterial genome dataset consumes 14.03% of the total processing time, whereas the polishing stage accounts for 61.88% of the time. In contrast, as depicted in Figure 2.3, the assembly of a human genome dataset occupies 59.74% of the total time, while polishing takes 20.89%. Despite these differences in time distribution across various datasets, polishing consistently ranks as either the first or second most time-intensive stage in the Flye workflow. Therefore, accelerating the polishing stage is crucial for enhancing the overall efficiency of the pipeline.

Polisher of Flye

Flye's polishing process begins by aligning all sequencing reads to the current assembly using minimap2-fast [9], an efficient aligner optimized for long-read sequences. Minimap2-fast is an improved version of the widely used minimap2 [13] software. Improvements are made through multiple optimizations using single-instruction multiple-data parallelization, efficient cache utilization, and a learned index data structure to accelerate seeding, chaining, and pairwise sequence alignment. These optimizations result in up to a 1.8-fold reduction in end-to-end mapping time while maintaining identical output. Minimap2-fast is a highly optimized tool, which is why it is used for alignment in the polishing stage of genome

assembly. The assembly is then divided into ”bubbles,” or smaller segments that highlight potential error regions. This segmentation allows Flye to perform targeted error correction. Within each bubble, Flye uses a maximum likelihood approach to assess the probability of different nucleotide sequences, selecting those that best fit the observed data. This systematic correction reduces errors and improves assembly quality. The polishing process can be repeated iteratively, leading to further improvements for some datasets. By refining errors through successive rounds of polishing, Flye is an effective tool for generating high-quality genomic assemblies, essential for advancing genomics research.

2.4. Sequence Alignment Algorithms

In the field of bioinformatics, sequence alignment is a fundamental technique used to identify regions of similarity between DNA, RNA, or protein sequences. Two widely used algorithms for this purpose are the Needleman-Wunsch [17] and Smith-Waterman [20] algorithms. Both algorithms utilize dynamic programming to perform sequence alignments, but they are designed for different types of alignment tasks.

Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm, introduced in 1970, is a global alignment algorithm. It aligns entire sequences from end to end and is particularly useful when the sequences are of similar length and expected to be homologous across their entire length. The algorithm constructs a scoring matrix where each cell represents the best alignment score up to that position. It fills the matrix using a recursive formula that considers matches, mismatches, and gaps, which allows the optimal alignment to be traced back from the last cell of the matrix.

Smith-Waterman Algorithm

In contrast, the Smith-Waterman algorithm, developed in 1981, is designed for local alignment. It identifies the optimal alignment between sub-sections of sequences, which is beneficial when dealing with sequences that may contain conserved regions interspersed with divergent or non-homologous segments. Like the Needleman-Wunsch algorithm, the Smith-Waterman algorithm uses dynamic programming to fill a scoring matrix. However, it differs in that negative scores are replaced with zeros, allowing the algorithm to identify the highest scoring sub-region within the matrix.

Comparison of Needleman-Wunsch and Smith-Waterman Algorithms

While both algorithms employ dynamic programming to compute sequence alignments, they have key differences that suit them to different types of alignment tasks. Both algorithms use a dynamic programming matrix to calculate alignment scores. They fill the matrix based on recursion relations involving scores for matches, mismatches, and gaps. Both algorithms can utilize customizable scoring matrices to determine match, mismatch, and gap penalties, which allows for flexible alignment based on specific biological contexts. The most significant difference is that Needleman-Wunsch performs global alignment, focusing on aligning entire sequences, whereas Smith-Waterman performs local alignment, targeting the most similar sub-sequences within the larger sequences. In Needleman-Wunsch, the scoring matrix is initialized to allow alignment from the beginning of the sequences, and the traceback is performed from the bottom-right corner to the top-left. In Smith-Waterman, the matrix is initialized to zero, and the traceback starts from the highest-scoring cell, allowing the identification of local matches. Needleman-Wunsch is best suited for comparing sequences of similar length that are believed to be homologous throughout their entirety. On the other hand, Smith-Waterman is advantageous when dealing with sequences of varying lengths or when conserved regions are expected to be embedded within non-homologous sequence contexts.

How Needleman-Wunsch Algorithm is Used for Flye’s Polisher

One of the steps in polishing involves correcting errors in each bubble using a maximum likelihood approach. This step takes a bubble as input, which includes read segments and a consensus sequence. The output is the consensus sequence that best represents all the read segments. To accomplish this, two scoring matrices are constructed for each pair of read segments and the consensus sequence using the Needleman-Wunsch algorithm: one forward and one backward. During this process, mutations such as deletions, insertions, or substitutions are introduced into the consensus sequence. The scores

for each read segment and the consensus sequence are calculated using these two matrices without needing to rerun the Needleman-Wunsch algorithm. The final score is determined by multiplying all the individual scores. If this final score is higher, the mutation is recorded. For example, when a deletion operation is introduced at any possible location in the consensus sequence, a comparison is made across all the higher scores, and only the sequence with the highest score is accepted. This process is repeated iteratively—building forward and backward scoring matrices and identifying the best single nucleotide mutation—until the consensus sequence no longer changes. At this point, the consensus sequence best represents all the read segments. The Needleman-Wunsch algorithm is a time-consuming process, and this operation is computed many times: the number of bubbles times the number of read segments for each bubble, multiplied by two. The computation is linear with respect to the genome length and the coverage. For human genome assembly, the large size of the sequence and the high coverage required for greater accuracy significantly increase the number of Needleman-Wunsch calculations compared to a small genome assembly.

2.5. Parallel Computing

BWA-MEM [12] is a widely used tool for mapping short reads to reference sequences due to its speed and accuracy. Various studies have accelerated BWA-MEM using different computing technologies. CPUs, with multithreading and SIMD capabilities, are commonly used to enhance performance through parallel processing, as seen in [22]. GPUs, known for their massive parallelism, have also been employed to speed up alignment tasks, as demonstrated by [8]. FPGAs offer custom hardware-level optimizations, providing significant speedups, as explored by [2]. This thesis will focus exclusively on optimizing Flye for CPUs, leveraging multithreading and SIMD for accessible and cost-effective improvements.

Parallel computing leverages advanced hardware architectures to boost software performance by executing multiple processes or operations simultaneously. Two crucial techniques in this domain are multi-threading and SIMD (Single Instruction, Multiple Data). Each technique utilizes hardware capabilities to optimize computational efficiency and speed. In practice, both multi-threading and SIMD are employed together to maximize computational efficiency. Multi-threading manages concurrent tasks across multiple cores, while SIMD accelerates data processing within each core. This combined approach leverages the strengths of both techniques, utilizing the full potential of modern processor architectures to achieve superior performance for complex and data-intensive applications.

Multi-threading

Multi-threading takes advantage of multi-core processors by dividing a single application into multiple threads that run concurrently. Each thread operates independently, allowing the processor to handle several tasks simultaneously. Modern processors are designed with multiple cores, each capable of executing its own thread. This parallelism is crucial for applications that perform numerous concurrent operations, such as web servers or complex simulations. By distributing tasks across multiple cores, multi-threading ensures that the processing power of each core is utilized effectively, leading to improved performance and responsiveness in software applications.

In the error correction process, the input consists of raw bubbles, while the output is polished bubbles. The process starts with an input bubble file that contains all the bubbles to be processed. Once a batch of bubbles is loaded from the file into memory, these bubbles can be distributed across multiple threads, allowing the polishing process to be executed in parallel. Multi-threading is an intuitive approach for this process since the error correction task can be parallelized by dividing the bubbles among different threads. However, several challenges must be addressed. For instance, the input file may be large, which can result in significant reading times. Additionally, the variability in bubble lengths can lead to inconsistent runtime across threads. Although the concept of using multi-threading for error correction is promising, the implementation requires careful consideration of these factors to ensure efficiency and effectiveness.

SIMD

SIMD enhances performance by utilizing specialized hardware instructions that perform the same operation on multiple data elements simultaneously. Processors equipped with SIMD capabilities, such as Intel's SSE or AVX instruction sets, are designed to handle large volumes of data efficiently. SIMD

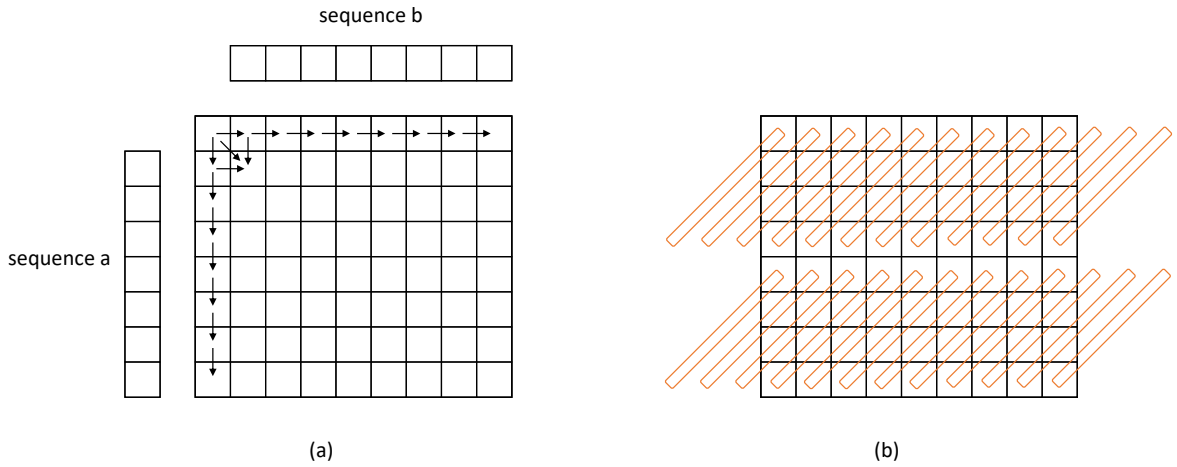


Figure 2.4: Image (a) displays the data dependencies within the Needleman-Wunsch algorithm and Image (b) shows the intra-vectorization approach for utilizing SIMD to optimize its performance.

instructions operate on data vectors, allowing a single instruction to process multiple data points in parallel. This is particularly beneficial for data-parallel tasks, such as multimedia processing or scientific computations, where the same operation needs to be applied to many data elements. SIMD instructions reduce the number of required operations and improve memory bandwidth utilization, resulting in faster and more efficient data processing.

In SIMD processing, two primary approaches are used for alignment: intra-vectorization and inter-vectorization. Each method has distinct advantages and disadvantages depending on the specific use case. In Figure 2.4(a), the data dependencies of the Needleman-Wunsch algorithm are illustrated. In the first column, each cell depends on the cell above it, while in the first row, each cell relies on the cell to its left. For all other cells, the value depends on the left, above, and diagonal (cross) cells, reflecting the dynamic programming approach used to compute alignment scores. Figure 2.4(b) demonstrates the intra-vectorization approach using SIMD operations to optimize performance. The data within the red rectangle can be accessed simultaneously, allowing these elements to be calculated together, thereby enhancing computational efficiency by processing multiple sequence elements concurrently. This approach parallelizes the calculation within an alignment process of a single read, making it particularly effective for scenarios requiring only one alignment at a time, such as database queries. It ensures that operations on individual alignment cells are handled concurrently, optimizing performance for single alignment tasks. Inter-vectorization, on the other hand, is suited for scenarios where multiple alignments need to be processed simultaneously. The efficiency of inter-vectorization depends on the number of reads and the specific SIMD instruction set used, as it allows for parallel processing of several alignment tasks. This method is especially useful when dealing with multiple alignments simultaneously, such as in error correction processes involving numerous segments. For error correction in a bubble, global alignment is required between each segment and the consensus sequence. In this context, each bubble contains multiple segments with similar lengths, minimizing unnecessary computations. Consequently, inter-vectorization proves advantageous for error correction, as it can efficiently handle multiple segments and alignments concurrently, reducing overhead and enhancing overall performance.

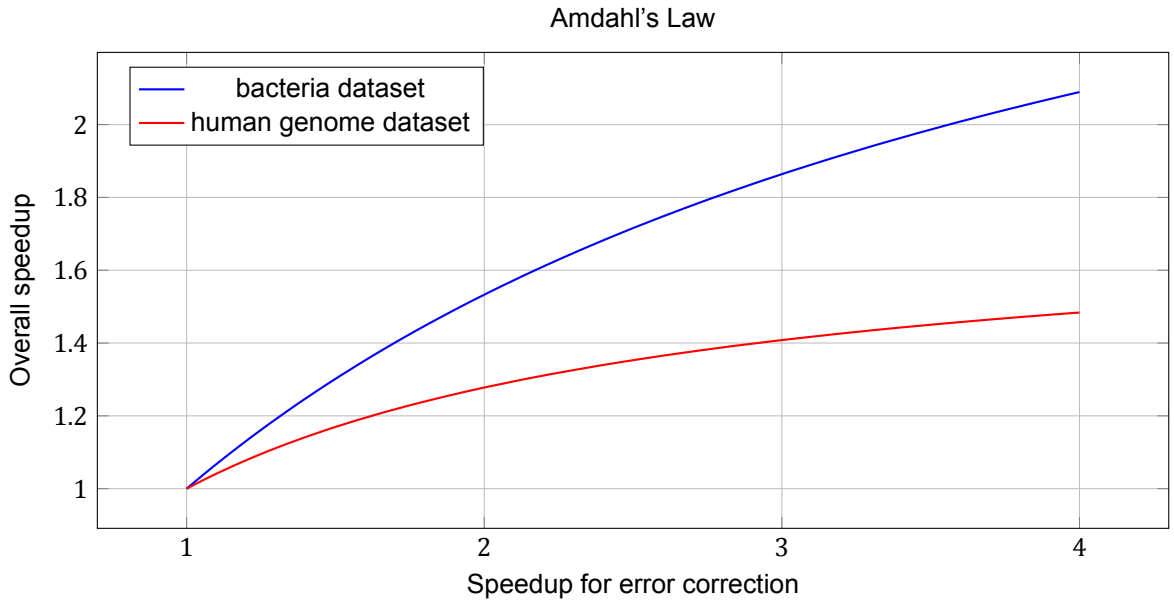


Figure 2.5: Amdahl's Law applied to the Flye's polisher, showing overall speedup versus speedup for error correction for two datasets.

2.6. Amdahl's Law

Amdahl's Law is a principle that highlights the potential speed-up of a computational process when only a portion of it is optimized. In the context of Flye's polishing step, which includes alignment, bubble splitting, and error correction, Amdahl's Law can help us understand the impact of improving each component. Flye's polisher incorporates Minimap2 for the alignment algorithm, a well-optimized tool known for its efficiency. The second component, splitting alignment into bubbles, is relatively quick and does not significantly contribute to processing time. Thus, the error correction phase is the primary candidate for optimization. The proportion of time dedicated to error correction varies with the dataset. For instance, in the bacteria dataset, error correction accounts for 69.52% of the total polishing time, whereas, in the human genome dataset, it constitutes 43.47%. This variation suggests that optimizing error correction can lead to substantial overall speed-ups, depending on the dataset. Figure 2.5 illustrates that optimizing error correction yields considerable improvements in processing speed for these datasets. AVX2, with its 256-bit data lanes capable of processing 4 data elements simultaneously, theoretically offers up to a 4x speedup for error correction. However, achieving this maximum speedup is impractical due to the overhead associated with implementing AVX instructions. The figure demonstrates the realistic speedups achieved for error correction, ranging from 1x to 4x.

3

Methods

3.1. General Polisher

3.1.1. Purpose and Implementation

Each bubble contains read segments $Segments = seg_1, seg_2, \ldots, seg_m$, and our objective is to determine a consensus sequence $Consensus$ that maximizes the probability $Pr(Segments|Consensus) = \prod_{i=1}^{m} Pr(seg_i|Consensus)$, where $Pr(seg_i|Consensus)$ represents the probability of producing segment seg_i from the consensus sequence $Consensus$. Given an alignment between a segment seg_i and the consensus $Consensus$, $Pr(seg_i|Consensus)$ is defined as the product of the match, mismatch, insertion, and deletion rates for all positions in this alignment. These rates should be derived from aligning any reads to a reference genome. A segment with a median length is selected from each bubble, and whether the consensus sequence for each bubble can be improved by introducing a single mutation in the selected segment is iteratively assessed. If a mutation is found that increases $Pr(Segments|Consensus)$, the mutation that results in the maximum increase is chosen and the process is repeated until convergence. The final sequence is then output as the error-corrected sequence of the bubble.

3.1.2. Initial Performance Enhancements

Before incorporating SIMD instructions, several optimizations were applied to enhance the general polisher's efficiency. Firstly, the intermediate vector used in the `substitution` (algorithm 4) and `insertion` (algorithm 5) functions was eliminated. This modification not only streamlined the code but also resulted in improved performance and reduced memory usage. Secondly, unnecessary initialization within the `getScoringMatrix` (algorithm 1) function was eliminated. Instead of initializing the entire 2D score matrix to zero, only the first element was set to zero. This optimization leverages the nature of the forward-backward dynamic programming algorithm, where subsequent matrix cells are computed using initialized values. This approach significantly reduces the initialization overhead, which is crucial given that initializing a large memory block to zero is time-consuming. The time complexity of this operation is proportional to $len(Consensus) \times len(seg_i)$, which becomes quadratic relative to the size of the reads, making it particularly inefficient for larger datasets. Thirdly, a new function, `getRevScoringMatrix` (algorithm 2), was introduced specifically to handle backward dynamic programming. Previously, the `getScoringMatrix` function was used for both forward and backward dynamic programming (by reversing the inputs seg_i and $Consensus$) to obtain the forward score matrix and backward score matrix. This dual-purpose approach complicated the implementation of the following `deletion` (algorithm 3), `substitution`, and `insertion` functions, especially when integrating AVX instructions later on. Figure 3.1 illustrates a `deletion` operation on the second element of $Consensus$. Figure 3.1(a) depicts the forward score indices, while Figure 3.1(b) shows the reverse score indices computed using the `getScoringMatrix` function. Figure 3.1(c) illustrates the reverse score indices using the `getRevScoringMatrix` function. The index of the reverse score increments by one in the row index, while the column index remains unchanged.

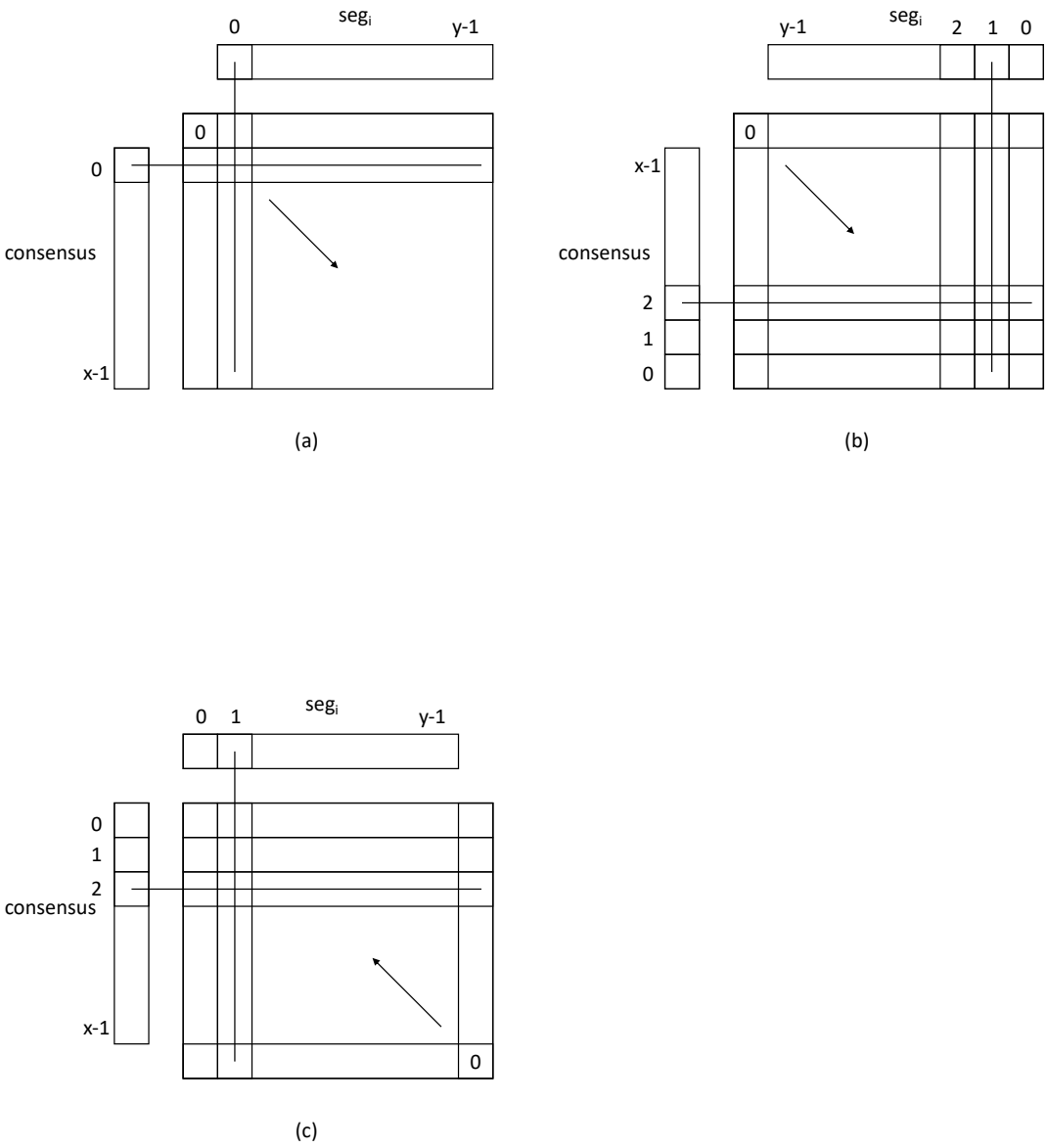


Figure 3.1: The calculation and the deletion operations of the forward and reverse score matrices. Image (a) displays the original forward matrix, Image (b) shows the original reverse matrix and Image (c) presents the enhanced reverse matrix.

Algorithm 1 getScoringMatrix

Require: Strings v, w; ScoreMatrix $scoreMat$
Ensure: Alignment score
Initialize $score \leftarrow 0$
for $i \leftarrow 0$ to $|v| - 1$ **do**
  $score \leftarrow getScore(v[i], '-')$
  $scoreMat[i+1, 0] \leftarrow scoreMat[i, 0] + score$
end for
for $i \leftarrow 0$ to $|w| - 1$ **do**
  $score \leftarrow getScore('-', w[i])$
  $scoreMat[0, i+1] \leftarrow scoreMat[0, i] + score$
end for
for $i \leftarrow 1$ to $|v|$ **do**
  $key1 \leftarrow v[i-1]$
  for $j \leftarrow 1$ to $|w|$ **do**
    $key2 \leftarrow w[j-1]$
    $left \leftarrow scoreMat[i, j-1] + getScore('-', key2)$
    $up \leftarrow scoreMat[i-1, j] + getScore(key1, '-')$
    $cross \leftarrow scoreMat[i-1, j-1] + getScore(key1, key2)$
    $score \leftarrow \max(left, up)$
    $score \leftarrow \max(score, cross)$
    $scoreMat[i, j] \leftarrow score$
  end for
end for
return $score$

Algorithm 2 getRevScoringMatrix

Require: Strings v, w; ScoreMatrix $scoreMat$
Ensure: Alignment score
Initialize $score \leftarrow 0$
for $i \leftarrow |v| - 1$ down to 0 **do**
  $score \leftarrow getScore(v[i], '-')$
  $scoreMat[i, |w|] \leftarrow scoreMat[i+1, |w|] + score$
end for
for $i \leftarrow |w| - 1$ down to 0 **do**
  $score \leftarrow getScore('-', w[i])$
  $scoreMat[|v|, i] \leftarrow scoreMat[|v|, i+1] + score$
end for
for $i \leftarrow |v|$ down to 1 **do**
  $key1 \leftarrow v[i-1]$
  for $j \leftarrow |w|$ down to 1 **do**
    $key2 \leftarrow w[j-1]$
    $right \leftarrow scoreMat[i-1, j] + getScore('-', key2)$
    $down \leftarrow scoreMat[i, j-1] + getScore(key1, '-')$
    $cross \leftarrow scoreMat[i, j] + getScore(key1, key2)$
    $score \leftarrow \max(right, down)$
    $score \leftarrow \max(score, cross)$
    $scoreMat[i-1, j-1] \leftarrow score$
  end for
end for
return $score$

3.1.3. SIMD Optimization

Implementing SIMD instructions for the general polisher involves several key steps: padding, constrained calculation, memory reorganization, and pre-computation.

Padding

Polishing requires performing the `alignment` operation on the $Consensus$ and each seg_i. We adopt inter-read vectorization to compute batches of score matrices instead of processing them one at a time. For each bubble, the $Segments$ are sorted according to their length. Padding is performed on the number of $Segments$ and the length of each seg_i. The general polisher calculates a 64-bit integer score. Taking AVX2 as an example, it can process 256 bits at a time, which equates to calculating four 64-bit scores simultaneously. Therefore, the batch size is set to four. As shown in In Figure 3.2(a), to handle cases where the number of $Segments$ is not a multiple of four, the last seg_m is duplicated multiple times to ensure there are no remaining segments when processing in batches of four. To accommodate varying lengths of seg_i, the length of the last seg_i in a batch is used as the target length for padding all other segments. In Figure 3.2(b), the colored cells represent meaningful values, while the uncolored cells serve to handle varying segment lengths and are not required for calculation. The score calculation necessitates careful constraints to ensure that only meaningful scores are computed.

Constrained calculation

Assume the length of $Segments$ is m. If m is a multiple of four, no padding is required. The final score is computed by adding the scores of four segments per batch and summing all intermediate scores. If m is not a multiple of four, the last segment seg_m needs to be duplicated $(4 - (m \mod 4))$ times. All batches, except the last one, are processed as above. For the last batch, only $(m \mod 4)$ scores are added to obtain the final score. Assume the batch index is b, and the first segment in batch b is seg_{4b}. Since the segments are sorted by ascending length, seg_{4b} is the shortest, allowing normal calculation up to its length. As the calculation progresses, the results should not overwrite the corresponding cell in the score matrix associated with seg_{4b}; instead, the value should remain the same as in the previous cell, necessitating an additional operation. A comparison mask, cmp_mask, is introduced to store the comparison result between the vector col and len. Here, col consists of four identical values of the current column, and len is a vector containing the lengths from seg_{4b} to seg_{4b+3}. This cmp_mask is utilized in a blending operation, determining whether the score should retain its previous value or adopt the newly calculated value based on cmp_mask. This constrained calculation ensures that the result remains consistent after padding.

Memory reorganization

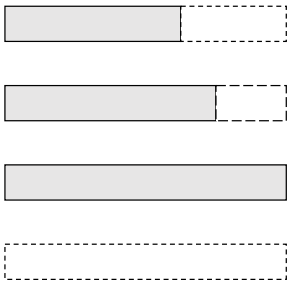
To optimize the memory access pattern for AVX2 instructions, the memory layout for the forward and reverse score matrices has been modified. The original layout consists of a continuous 1D memory chunk representing a 2D score matrix for each $Consence$ (row) and seg_i (column). The modified memory layout remains 1D but now represents a forward and reverse 3D matrix for each $Consence$ and a batch of seg values (from seg_{4b} to seg_{4b+3}). This transformation effectively introduces a depth dimension, converting the original 2D matrix into a 3D matrix. In the 2D matrix layout, the column values are stored contiguously. In contrast, the 3D matrix layout stores the depth values contiguously. Specifically, for a 2D matrix, the memory location of a cell (i, j) is calculated as:

$$\text{index}(i, j) = i \times \text{cols} + j$$

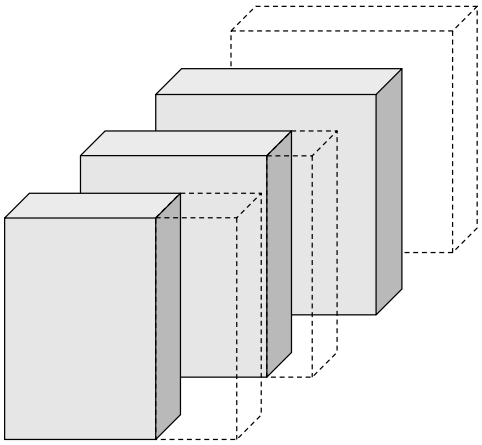
For the 3D matrix, the computation of four cells $(i, j, 0)$ to $(i, j, 3)$ is performed simultaneously, with their memory locations calculated as:

$$\text{index}(i, j, d) = i \times \text{cols} \times 4 + j \times 4 + d \quad \text{for} \quad d = 0, 1, 2, 3$$

This layout modification ensures that the data is contiguous in memory when accessing the depth dimension, which enhances the efficiency of SIMD operations. By computing multiple depth values simultaneously, we reduce cache misses and improve the algorithm's overall performance.



(a) Depicts the padding process for reads in the final batch. The gray rectangle represents the original reads, while the dashed rectangle indicates the padded areas. Two types of padding are applied: first, each read is padded to match the length of the longest read in the batch; second, the total number of reads is padded to be a multiple of the batch size.



(b) Shows the score matrix for the final batch. The gray area represents meaningful data, while the transparent area indicates padding used for calculation purposes. This padding is necessary but not relevant to the final results.

Figure 3.2: The padding process for reads and score matrix in AVX instruction calculations.

pre-computation
The probabilities for match, miss, delete, and insertion events can be found in Appendix A.1. The function `probToScore` computes the alignment score from a given probability p using the formula:

$$\text{score} = \text{round}(\ln(p) \times 131072)$$

where $\ln(p)$ is the natural logarithm of the probability p, and the constant 131072 is equivalent to 2^{17}.

To facilitate rapid score lookups, a substitution matrix is implemented. Although a 5×5 matrix would suffice, a 256×256 matrix is utilized for efficiency. This design allows for direct indexing using ASCII values. For example, to retrieve the score for a match involving character A (ASCII value 65), the index is computed as $65 \times 256 + 65$. In the original implementation, substitution values are computed during runtime. However, in the SIMD-optimized version, all values are pre-calculated. This pre-computation enables direct loading of values into AVX registers, thereby eliminating computational delays. Data storage is optimized for cache efficiency. For each batch of operations, five 2D matrices are maintained. Each matrix corresponds to the scores from a character to segments $\text{seg}_i, \text{seg}_{i+1}, \text{seg}_{i+2}, \text{seg}_{i+3}$. Consequently, the dimensions of each matrix are $\text{len}(\text{seg}_{i+3}) \times$ batch size. These pre-calculated matrices are utilized across the `alignment`, `insertion`, and `substitution` functions, ensuring efficient reuse of computed values and significantly reducing the need for recalculations.

3.2. Dinucleotide Fixer

3.2.1. Purpose and Implementation

The `DinucleotideFixer` class is designed to optimize sequences by adjusting dinucleotide runs within a candidate sequence. Its primary purpose is to refine the sequence within a structure referred to as a `bubble`, ensuring that it aligns more closely with a set of provided reference sequences, or `branches`. The `fixBubble` method, central to this class, assesses the quality of the alignment of a candidate sequence by evaluating the likelihood scores derived from a global alignment process. If the sequence contains a long run of repeated dinucleotides, the method attempts to modify the length of these runs by either inserting or deleting dinucleotide pairs. The class employs the helper method `getDinucleotideRuns` to identify the position and length of the longest run of repeated dinucleotides. If either increasing or decreasing the dinucleotide run improves the alignment score compared to the original sequence, the sequence is updated, and this change is logged for potential polishing steps. This process allows the `DinucleotideFixer` class to enhance the structural integrity and biological plausibility of the sequence by fine-tuning repetitive dinucleotide motifs.

3.2.2. Separate Alignment Class for Dinucleotide Fixer

In contrast to the `alignment` function used in the `GeneralPolisher`, which employs both forward and backward dynamic programming processes and retains a score matrix for subsequent use, the `DinucleotideFixer` requires only the final alignment score. The original `alignment` function, optimized with SIMD for performance, is designed for comprehensive alignment tasks, which involves additional memory overhead and computation. To address this, a specialized `alignment` function was developed specifically for the `DinucleotideFixer`. This new implementation performs a single forward dynamic programming pass, thereby streamlining the calculation process and significantly reducing memory usage. This approach enhances both computational efficiency and memory management, tailored to the specific needs of dinucleotide run adjustment.

3.3. Bubble Processor

3.3.1. Purpose and Implementation

In the previous section, a general polisher was introduced, which polishes a single bubble at a time. This section describes the `bubble processor`, a tool designed to read input data from a file, polish bubbles using multiple threads, and write the processed data to an output file. For small datasets, these three tasks can be executed sequentially. However, as the dataset size increases, reading data from the file can become time-consuming, and loading all data into memory can lead to high memory consumption. To address these challenges, a multi-threaded architecture (Figure 3.3) is employed.

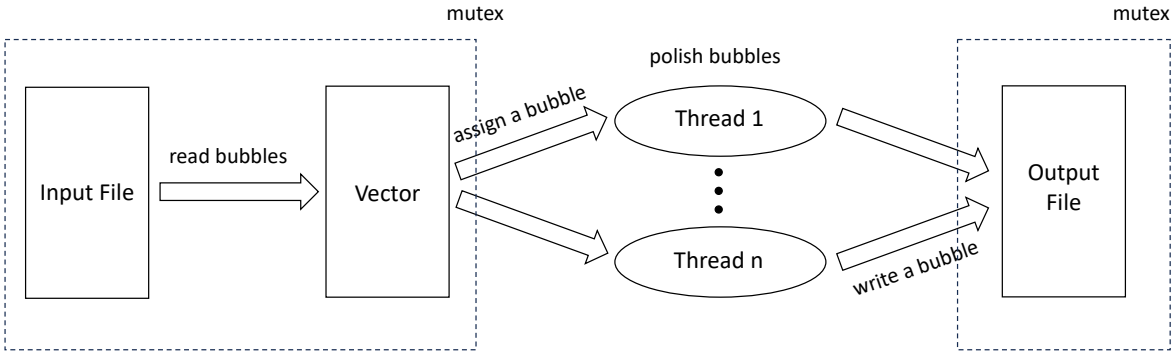


Figure 3.3: The original multi-threaded architecture (Design 1).

This approach involves several key steps. First, a default batch size of 100 is used to read data from the file. This limits the amount of data loaded into memory at any given time, reducing memory consumption. Second, after a batch of data is read into memory, multiple threads are deployed to process the bubbles. Each thread attempts to retrieve a bubble from memory, polish it, and write the output to a single file. This process continues until all bubbles in the batch are processed. Third, the first thread to complete its work starts reading the next batch of data. This process repeats until all bubbles in the file are polished. By overlapping data reading with bubble polishing, part of the data reading time is effectively hidden behind the processing time, thereby reducing the total runtime. However, the implementation relies heavily on mutexes for managing concurrent access to memory and the output file. When all threads try to retrieve a bubble or write to the file simultaneously, it leads to notable waiting times due to mutex contention.

3.3.2. New Multi-threaded Architectures

In the original design, bubble polishing was performed using multiple threads. However, the polished bubbles were written to a single output file, necessitating the use of a mutex to prevent data races. This approach led to significant contention and waiting times due to frequent mutex acquisition. Our new approach (`design_2`, Figure 3.4) addresses this issue by binding each thread to a separate output file. This allows the polished bubbles to be written concurrently without the need for mutexes, thus eliminating the associated contention. Furthermore, we have optimized the processing of bubbles by introducing batch processing. Instead of assigning a single bubble to each thread at a time, a batch of bubbles is now allocated to each thread. This reduces the frequency of mutex acquisitions, as the mutex is only acquired once per batch, rather than once per bubble. Consequently, the mutex acquisition time is approximately inversely proportional to the batch size. A larger batch size reduces the total mutex acquisition time, thereby shortening the waiting period for threads. However, choosing an appropriate batch size is crucial. If the batch size is too large, the reading, polishing, and writing operations might be executed sequentially, negating the benefit of hiding reading time behind polishing. Therefore, the batch size must be carefully determined to balance these factors. Additionally, the number of bubbles varies across datasets, influencing the optimal batch size. The choice of batch size must consider the specific characteristics of each dataset to maximize performance.

To enhance the efficiency of our multi-threaded architecture, we introduced `design_3` (Figure 3.5).

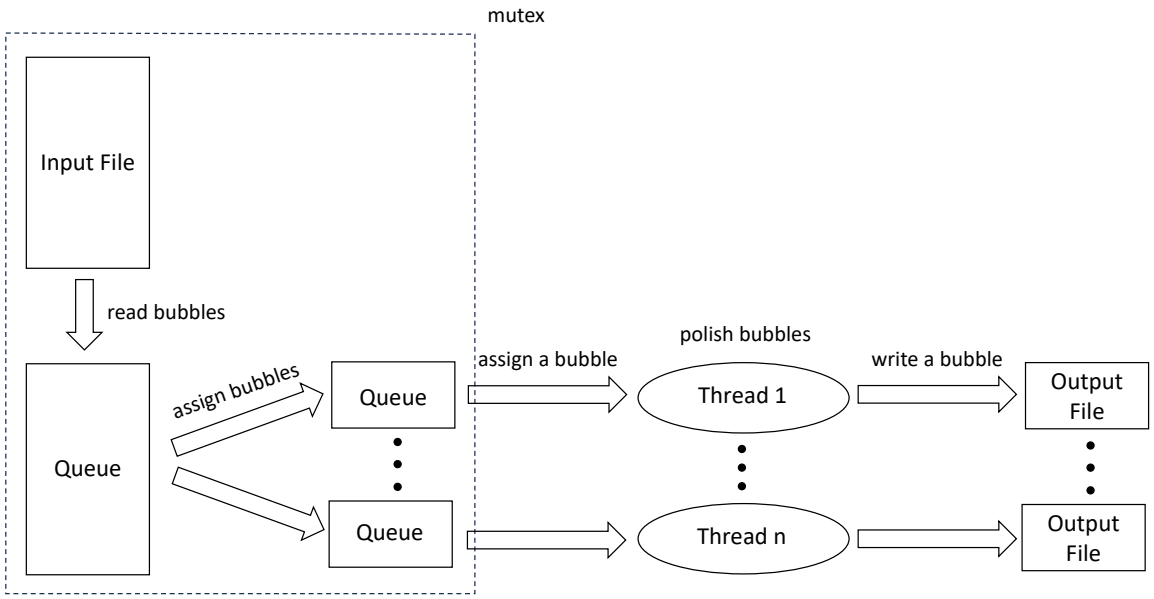


Figure 3.4: The improved multi-threaded architecture (Design 2), featuring enhancements such as writing to separate files and utilizing batch processing for increased efficiency.

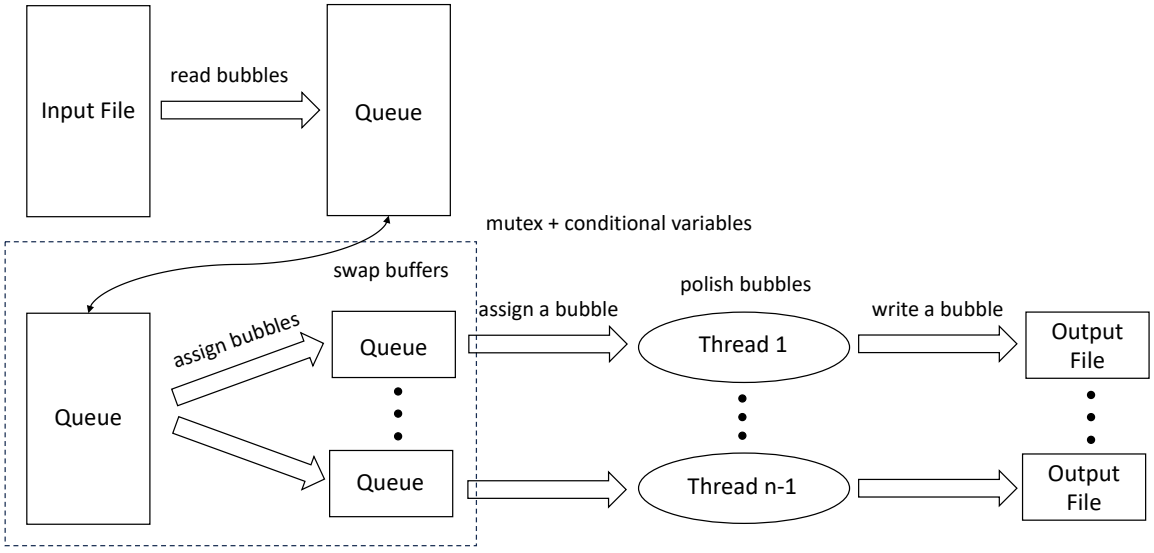


Figure 3.5: The improved multi-threaded architecture (Design 3) designed for handling a large number of threads, incorporating a dedicated thread for file reading and employing double buffering for enhanced performance.

Unlike `design_2`, `design_3` incorporates an additional component, the `bubble processor pro`, specifically designed to handle situations involving a large number of threads. In scenarios with a small number of threads, the standard `bubble processor` is utilized. In this configuration, all threads are dedicated to polishing tasks. Once a thread completes its current task, it retrieves the next batch of bubbles for processing. After the retrieval process, all threads work together to polish the new batch of bubbles. When the number of threads is large, the `bubble processor pro` comes into play. This version reserves one thread exclusively for reading tasks, while the remaining threads focus on polishing. This setup allows reading and polishing processes to occur simultaneously, reducing idle time as reading tasks are masked by the polishing operations. However, this approach does sacrifice one thread for polishing, making it only advantageous when the number of threads is sufficiently large to justify the trade-off. Additionally, `bubble processor pro` implements double buffering. This technique enables simultaneous operations: one buffer is used for reading bubbles from a file, while the other distributes bubbles to threads for processing. When the second buffer is depleted and the first buffer is replenished, the buffers swap roles, allowing continuous processing and reading without interruptions.

4

Experiments

4.1. Experimental Setup

The experiments were conducted using two distinct server clusters. The first server is equipped with an Intel Xeon Silver 4114 CPU, which supports AVX2 and AVX512 instruction sets, 96 GB of RAM, and 1 TB of local storage. The second server features a AMD EPYC 7532, which supports only the AVX2 instruction set, 1 TB of RAM, and 1 TB of local storage. The detail of Intel Xeon Silver 4114 and AMD EPYC 7532 CPUs can be found in Table 4.1. The optimized polisher was evaluated on two datasets: the E. coli P6-C4 PacBio data (bacterial dataset) and the HG002 (NA24385) Oxford Nanopore Ultra-long GridION data (human genome dataset). Due to the memory limitations of the first server, the polisher was tested on the HG002 dataset exclusively on the second server. The polisher was tested on the E. coli dataset not only on the first server to observe the effects of AVX512 optimizations but also on the second server to see the effects of high parallelism levels.

	Intel Xeon Silver 4114	**AMD EPYC 7532**
Sockets × Cores × Threads	2 × 10 × 2	2 × 32 × 2
AVX register width (bits)	512, 256, 128	256, 128
Vector Processing Units (VPU)	2/Core	2/Core
Base Clock Frequency (GHz)	2.20	2.40
L1D Cache (KB)	32	32
L2 Cache (KB)	1024	512
L3 Cache (MB) / Socket	13.75	256

Table 4.1: Details of Intel Xeon Silver 4114 and AMD EPYC 7532 CPUs

Figure 4.2 shows the description of various designs. `baseline` refers to the original multi-thread architecture, serving as the baseline for our comparisons. `design_2` and `design_3`, introduced in the Method section, represent the new multi-thread architectures. Each architecture variant is further optimized using different SIMD instructions, denoted by `avx2` and `avx512`. These optimizations are applied to the `GeneralPolisher` and `DinucleotideFixer` components. For instance, `baseline` represents the baseline architecture without additional SIMD optimization, while `avx2_multithread_opt` refers to the new multi-thread architecture with AVX2 optimization.

Design Name	**Description**
`baseline`	The baseline for our comparisons.
`avx2_opt`	The baseline architecture with AVX2 optimization.
`avx512_opt`	The baseline architecture with AVX-512 optimization.
`avx2_multithread_opt`	The new multi-thread architecture with AVX2 optimization.

Table 4.2: Description of various designs and their SIMD optimizations.

Bacteria Dataset	baselinse	avx2_opt	avx512_opt
Profiling flye			
assembly	05:12	-	-
consensus	07:32	-	-
repeat	01:19	-	-
contigger	00:02	-	-
polishing	22:59	-	-
Profiling polishing			
minimap2	01:53	-	-
make bubbles	05:05	-	-
correct errors	15:55	5:17	3:39

Table 4.3: Profiling Data (mm:ss) for `Flye` and `Polishing` components on the bacteria dataset running with one thread: baseline vs avx2_opt vs avx512_opt

Component	baseline	avx2_opt	avx512_opt
BubbleProcessor (Total: 947.98 / 317.18 / 219.36)			
GeneralPolisher	809.50	265.63	177.49
DinucFixer	76.59	10.94	6.12
Other	61.89	40.61	35.75
GeneralPolisher			
Alignment	308.81	143.14	81.39
Deletion	43.97	32.30	33.35
Insertion	299.04	53.98	35.79
Substitution	148.71	26.92	18.68
Other	8.97	60.84	50.15

Table 4.4: Profiling Data (seconds) for `BubbleProcessor` and `GeneralPolisher` components on the bacteria dataset running with one thread: baseline vs avx2_opt vs avx512_opt

4.2. Flye Profile

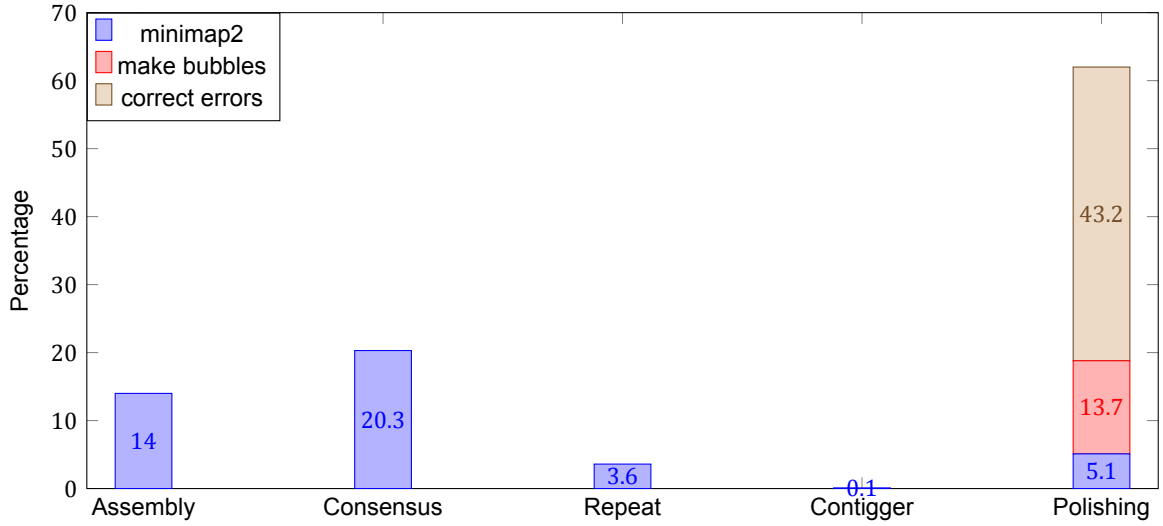
4.2.1. Bateria Dataset

The Flye assembler was executed on a bacterial dataset using a single-threaded configuration. Table 4.3 shows the profiling results for the bacteria dataset. Analysis of the runtime distribution (Figure 4.1a) revealed that the polishing stage constitutes 61.88% of the total processing time. Within the polishing stage, the breakdown of time allocation is as follows: minimap2 accounts for 8.24%, creating bubbles takes up 22.24%, and error correction comprises 69.52% of the time. These results indicate that the polishing stage is the most time-consuming component of the assembly process. Furthermore, error correction, which is the primary focus of this thesis, emerges as the most time-intensive sub-process within the polishing stage.

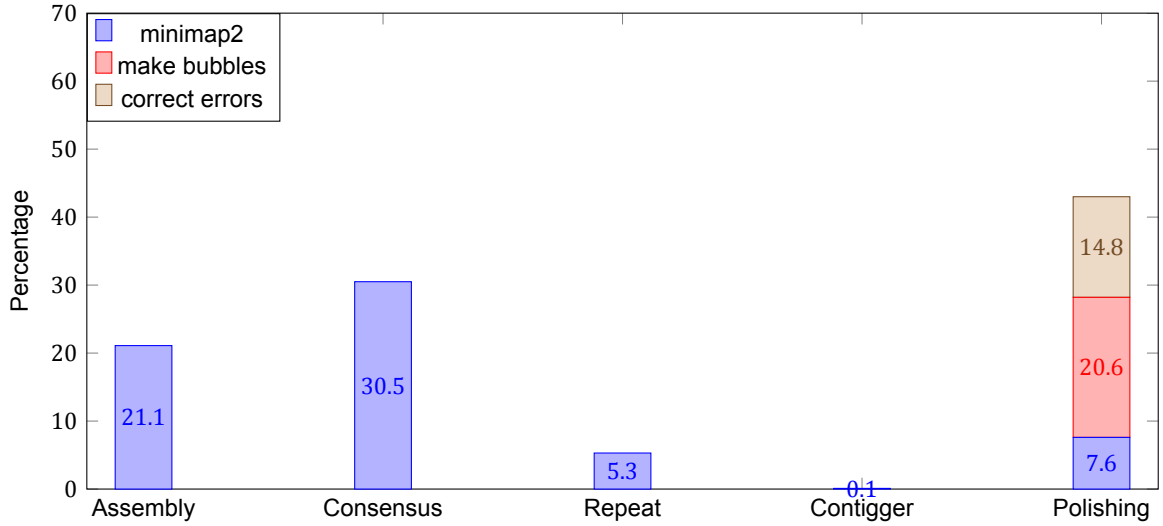
Further profiling was conducted on the error correction component of the polishing stage. As detailed in Table 4.4, the `BubblesProcessor`, which handles error correction, requires a total of 947.98 seconds. Among its sub-components, `GeneralPolisher` is the most time-consuming, accounting for 809.50 seconds, or approximately 85.39% of the total processing time. This indicates that optimizing `GeneralPolisher` could substantially decrease overall processing time. The `DinucFixer` accounts for 8.08% of the total time, making it the second largest contributor. The remaining time is attributed to I/O operations and multi-threading overhead. Within `GeneralPolisher`, the alignment and insertion processes are the most time-intensive, representing 75.09% of its total processing time.

4.2.2. Human Genome Dataset

The Flye assembler was executed on an HG002 human genome dataset using all 64 threads (32 cores) on a single socket. Table 4.3 shows the profiling results for the human genome dataset. As shown in Figure 4.2a, the polishing stage accounted for 20.89% of the total processing time, with the



(a) Baseline

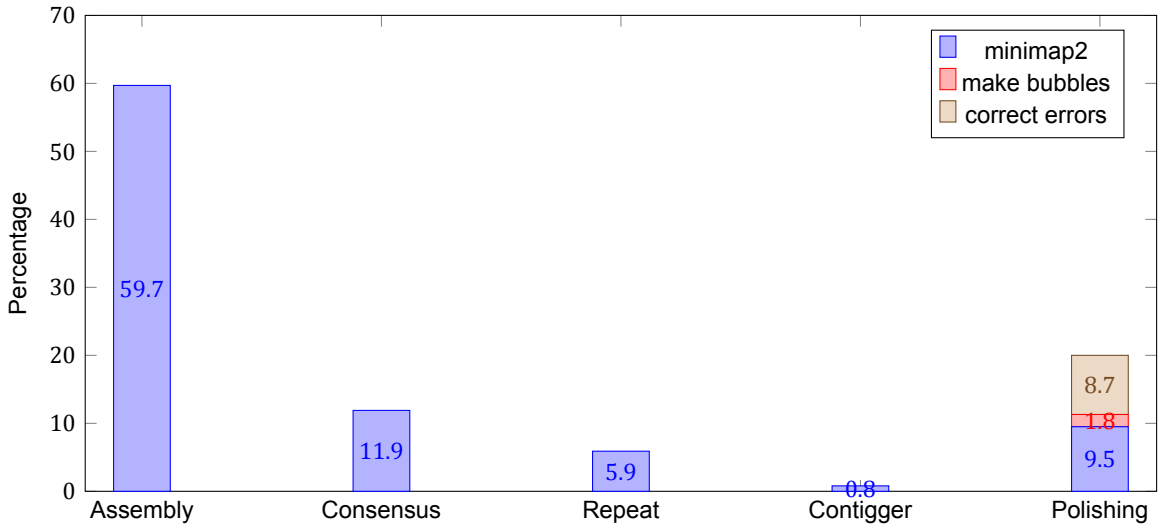


(b) avx512_opt

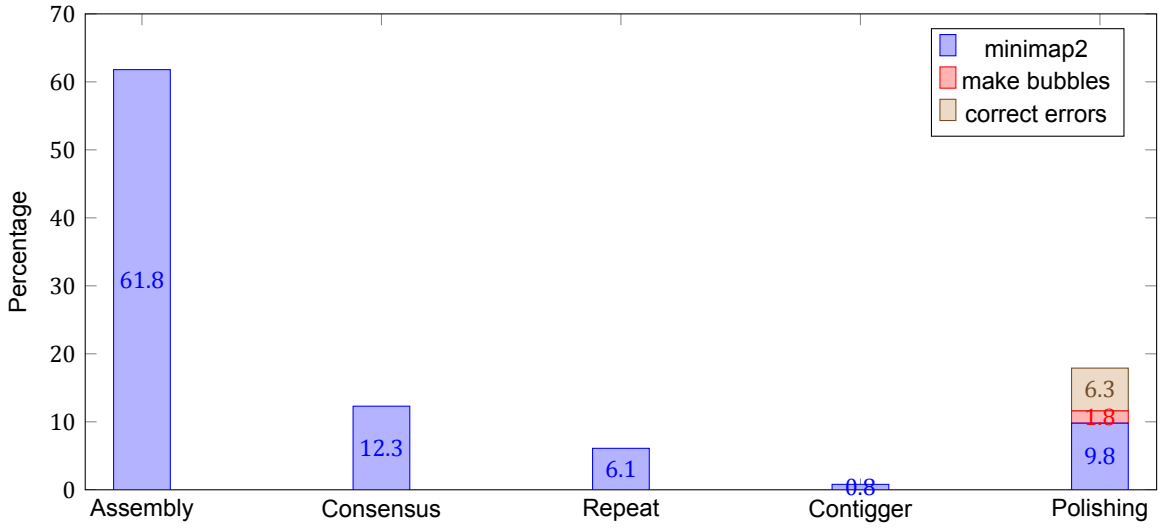
Figure 4.1: Time distribution across different stages of Flye assembler on bacteria dataset with detailed polishing steps before and after acceleration running with one thread.

Human Genome Dataset	baseline	avx2_multithread_opt
Profiling flye		
assembly	2-08:08:46	-
consensus	0-11:08:39	-
repeat	0-05:32:53	-
contigger	0-00:45:09	-
polishing	0-19:37:49	-
Profiling polishing		
minimap2	0-08:52:51	-
make bubbles	0-01:38:25	-
correct errors	0-08:05:29	0-5:52:43

Table 4.5: Profiling Data (d-hh:mm:ss) for `Flye` and `Polishing` components on the human genome dataset running with 64 threads: baseline and avx2_multithread_opt



(a) Baseline



(b) avx2_multithread_opt

Figure 4.2: Time distribution across different stages of Flye assembler on the human genome dataset running with 64 threads with detailed polishing steps before and after acceleration.

Component	baseline	avx2_opt	avx2_multithread_opt
BubbleProcessor (Total: 28679.28 / 25454.06 / 20713.96)			
GeneralPolisher	25255.13	13791.92	18044.59
Waiting	2370.28	11276.43	2175.38
DinucFixer	880.62	164.05	312.84
Other	173.25	221.66	181.15
GeneralPolisher			
Alignment	10059.64	5829.22	6865.57
Deletion	1349.87	2796.77	3339.32
Insertion	10786.70	4119.11	5805.63
Substitution	3014.19	870.35	1364.64
Other	44.73	176.47	669.43

Table 4.6: Profiling Data (seconds) for `BubbleProcessor` and `GeneralPolisher` components on the human genome dataset running with 64 threads: baseline, avx2_opt and avx2_multithread_opt

error correction process within this stage consuming 43.47% of that time. Although the polishing stage is no longer the most time-consuming part for the human genome dataset, it still requires nearly 20 hours to complete. The Flye polisher also functions as a versatile standalone tool, capable of refining existing assemblies produced by other assemblers. This feature allows researchers to integrate Flye's advanced polishing algorithms into various workflows, enhancing the accuracy and quality of genomic assemblies across different projects. As the final stage in the assembly process, polishing is crucial for ensuring accuracy. Flye executes a single polishing iteration by default, but additional iterations can correct a small number of residual errors, potentially improving overall assembly quality. However, these extra iterations increase processing time, making the optimization of the Flye polisher essential as it enables researchers to achieve high-quality assemblies efficiently, balancing the trade-off between accuracy and computation time.

To further investigate the error correction process during the polishing stage, we analyze the time distribution shown in Table 4.6. The `GeneralPolisher` sub-process dominates the time expenditure, constituting approximately 88.06% of the total execution time for the `BubbleProcessor`. Within `GeneralPolisher`, the alignment task requires about 35.08% of the total time, while the insertion task takes roughly 37.61%. In contrast, the `dinucFixer` sub-process accounts for only about 3.07% of the total processing time. Additionally, the wait time represents 8.26% of the `BubbleProcessor` duration. The profiling results indicate that the primary bottleneck is within the `GeneralPolisher`, specifically due to the time-intensive alignment and insertion tasks, which together account for over 70% of the total processing time. Notably, the `dinucFixer` is no longer the second most time-consuming component; instead, the wait time has become a more significant factor compared to the results obtained from the bacteria dataset.

4.3. Performance Comparison

4.3.1. Bacteria Dataset

In this experiment, we assembled the bacterial dataset using a single thread, comparing the baseline and a optimized versions of the architecture. Figure 4.1 shows the time distribution across different stages of Flye assembler on the bacteria dataset before (Figure 4.1a) and after acceleration (Figure 4.1b). The error correction process during the polishing stage was previously the most significant bottleneck in the Flye workflow, accounting for 42.92% of the total processing time. However, after implementing acceleration techniques, the time required for this stage has been significantly reduced, now comprising only 14.78% of the overall processing time. Consequently, it has shifted to become the fourth most time-consuming component in the Flye workflow.

The optimizations significantly enhanced the performance of the `GeneralPolisher` and `DinucleotideFixer` components. For the `BubblesProcessor`, the total execution time decreased from 947.98 seconds to 317.18 seconds with AVX2 optimization and further to 219.36 seconds with AVX512 optimization, achieving an overall speedup of approximately 4.32 times. Within the `GeneralPolisher` component, execution time was reduced from 809.50 seconds to 265.63 seconds with AVX2

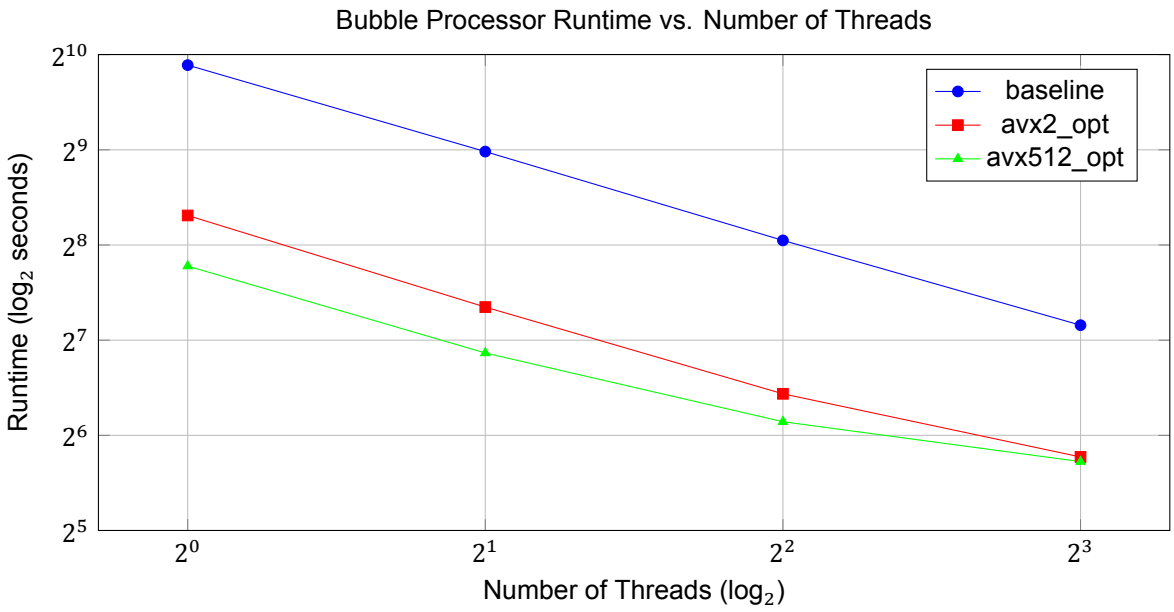


Figure 4.3: Runtime comparison across different threading ($\log_2$ scale for both axes): baseline vs avx2_opt vs avx512_opt

and then to 177.49 seconds with AVX512, yielding a combined speedup of about 4.56 times. The `DinucleotideFixer` component saw its execution time drop from 76.59 seconds to 10.94 seconds with AVX2 and further to 6.12 seconds with AVX512, resulting in a substantial speedup of approximately 12.51 times.

Figure 4.3 illustrates the runtime performance of the `BubbleProcessor` as the number of threads increases, comparing three different implementations: Baseline, AVX2, and AVX512. The graph uses a $\log_2$ scale for both the number of threads and the runtime, enabling a clear view of the performance trends across exponential increases in these variables. As expected, the runtime decreases with an increasing number of threads for all three implementations, demonstrating the effectiveness of parallelization. The Baseline implementation shows the highest runtimes across all thread counts, with a runtime of 947.98 seconds for a single thread, reducing to 142.67 seconds at eight threads. This represents a speedup of approximately 6.64 times. The AVX2 implementation improves upon the Baseline, starting at 317.18 seconds with one thread and dropping to 54.65 seconds with eight threads, resulting in a speedup of about 5.80 times. The AVX512 implementation further enhances performance, starting at 219.36 seconds with one thread and reducing to 52.83 seconds with eight threads, achieving a speedup of around 4.15 times. Notably, the performance gains between the AVX2 and AVX512 implementations narrow as the number of threads increases, highlighting diminishing returns from additional vectorization optimizations at higher parallelism levels.

4.3.2. Human Genome Dataset

In this experiment, we assembled the human genome dataset using 64 threads (32 cores) on a single socket, comparing the baseline and a optimized versions of the architecture. Figure 4.2 shows the time distribution across different stages of Flye assembler on the human genome dataset before (Figure 4.2a) and after acceleration (Figure 4.2b). Figure 4.4 presents a comparison of speedups between avx2_opt and avx2_multithread_opt relative to the baseline. The experimental results demonstrate that during the entire polishing stage, the avx2_opt implementation achieves a speedup of 1.12x, while avx2_multithread_opt shows a more significant improvement with a speedup of 1.21x, reducing the overall processing time by 2 hours and 20 minutes compared to the original polisher. When focusing on specific components, the `BubbleProcessor` exhibits a speedup of 1.1x with avx2_opt, whereas avx2_multithread_opt achieves a markedly higher speedup of 1.4x. In contrast, for the `GeneralPolisher`, avx2_opt outperforms avx2_multithread_opt with a speedup of 1.8x compared to 1.4x. To further investigate the causes of these performance variations, a detailed profiling of the subcomponents was conducted, yielding valuable insights into the observed differences.

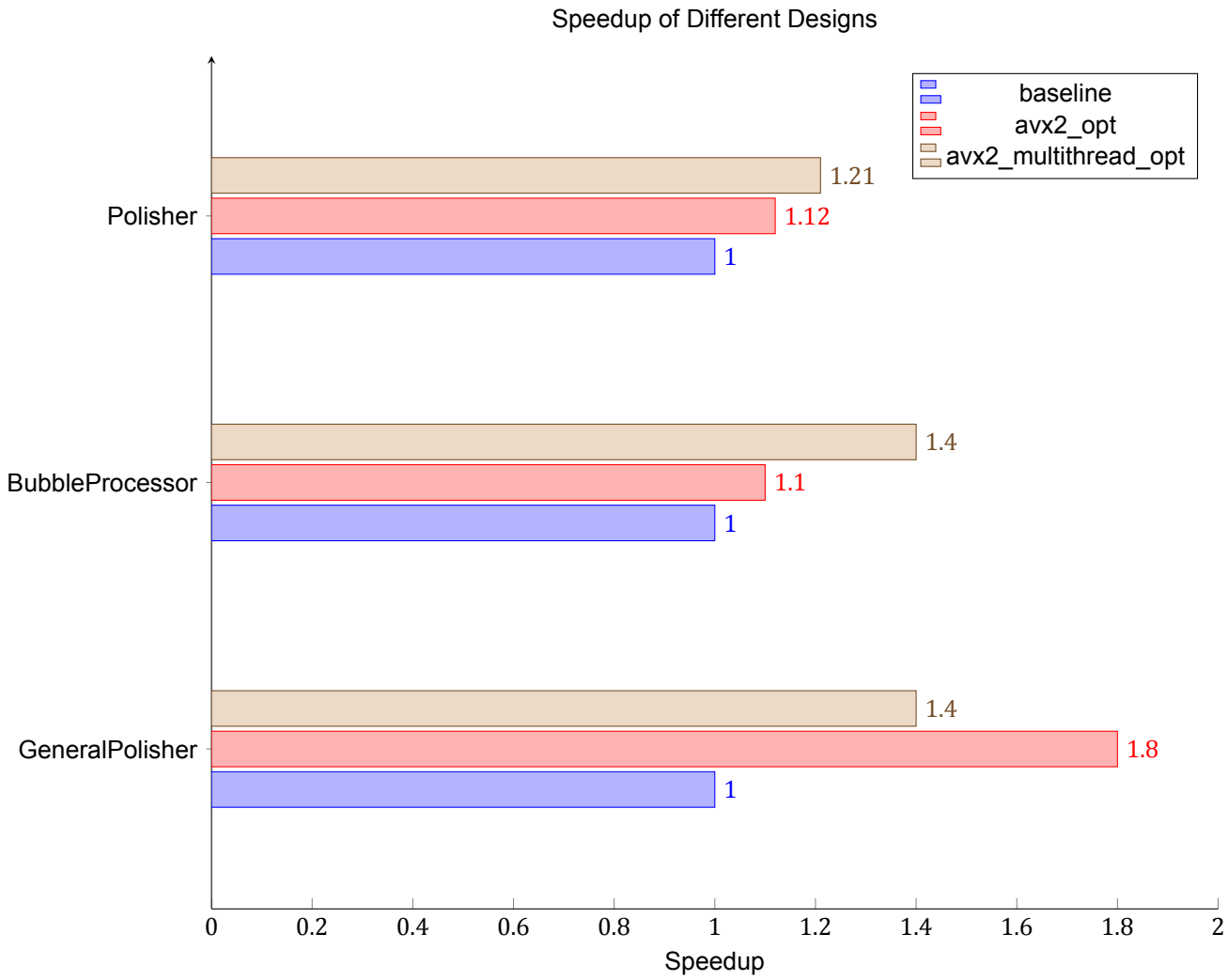
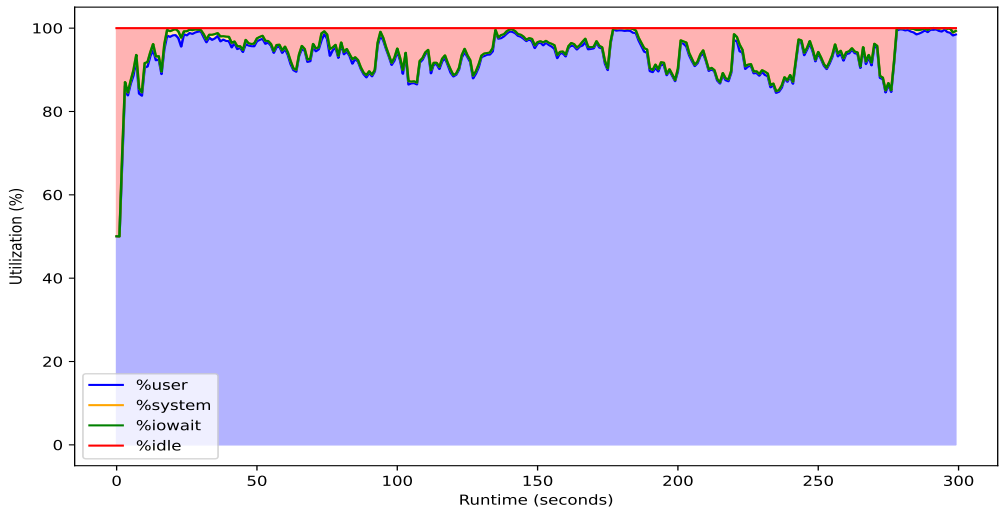


Figure 4.4: Comparison of speedups for avx2_opt and avx2_multithread_opt against the baseline on the human genome dataset running with 64 threads.

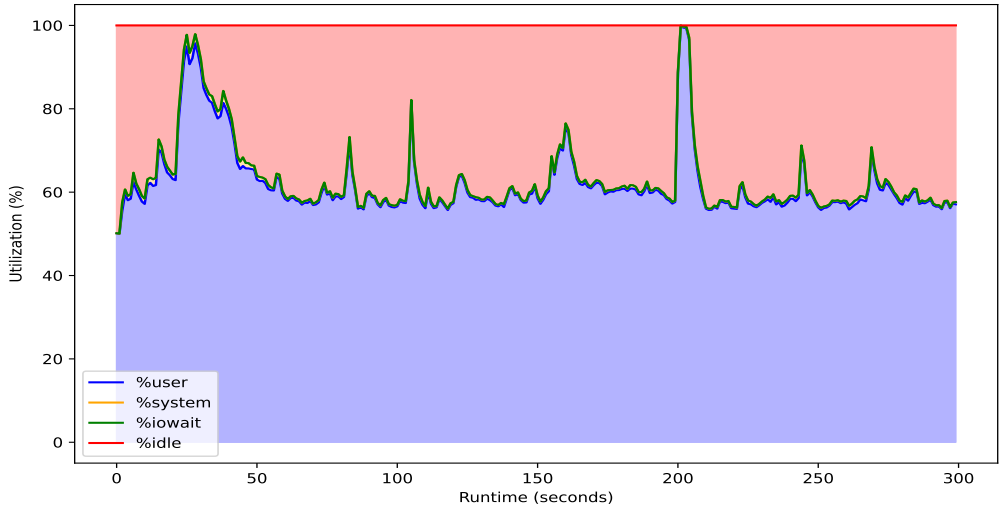
Table 4.6 presents a comparative analysis of the `BubbleProcessor` execution times, highlighting a reduction in total processing time with the `avx2_opt` configuration. This design achieves an 11.25% increase in speed over the baseline. Key improvements in `avx2_opt` include a substantial decrease in the `GeneralPolisher` processing time, from 25,255.13 seconds to 13,791.92 seconds, yielding a 1.8-fold speedup due to the enhanced performance provided by AVX2 optimizations. Moreover, the `DinucleotideFixer` process experiences a significant reduction in execution time, dropping from 880.62 seconds to 164.05 seconds. This demonstrates the efficacy of the optimizations applied in this section of the pipeline. However, it is important to note the increase in wait times, indicating a bottleneck in the original multi-threaded architecture. Consequently, despite the substantial decrease in computation time, the overall reduction in processing time is modest, primarily due to these increased wait times. This analysis suggests that further optimization of the data access and scheduling strategies is necessary to fully capitalize on the computational enhancements. We introduced a new multithreaded architecture that writes to separate files and employs batch processing to reduce waiting time. The runtime results are under the `avx2_multithread_opt` configuration. This change reduced the waiting time significantly, from 11,276.43 seconds to 2,175.38 seconds. Although the processing time for `GeneralPolisher` and `DinucleotideFixer` increased slightly, the overall runtime is shorter compared to the `avx2_opt` configuration. This adjustment achieves a 1.4-fold speedup over the baseline.

Figure 4.5 illustrates the CPU utilization over time for three different settings, each utilizing 64 threads (32 physical cores). Due to the long runtime of the experiments, the utilization data is only shown for a fixed, short period to represent the overall process. In Figure 4.5a, the CPU utilization is consistently high, indicating that all cores are actively engaged in polishing the bubbles. In contrast, Figure 4.5b shows a significant drop in average CPU utilization. Many cores remain idle because the time required for polishing is significantly reduced, making it comparable to the time needed for reading bubbles. Consequently, cores that have completed their work must wait for bubble reading, leading to lower CPU utilization. This observation correlates with the runtime results for `avx2_opt`, which shows significant improvement in polishing time but only a modest overall runtime reduction due to increased waiting time. Conversely, Figure 4.5c demonstrates high CPU utilization, comparable to that of baseline. This indicates that the enhanced multithreaded architecture effectively reduces waiting time. In this scenario, the reading process can lag behind the polishing process, ensuring that the CPU resources are fully utilized.

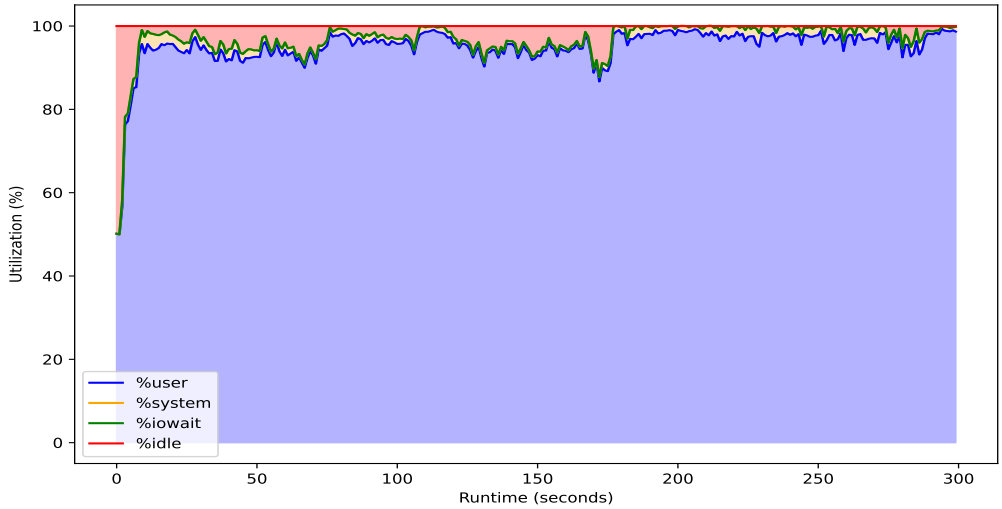
The experiment demonstrates that `avx2_multithread_opt` consistently outperforms baseline when evaluated with both 64 threads (utilizing hyper-threading) and 32 threads (using only physical cores). This section extends the comparison of `avx2_multithread_opt` and baseline to a broader range of thread counts: 1, 2, 4, 8, and 16. In this context, a key assumption is that doubling the number of threads should ideally halve the runtime. However, running with a low thread count results in extremely prolonged execution times. To mitigate this, the experiment employs 1 million bubbles, significantly reducing the total execution time. As illustrated in Figure 4.6, `avx2_multithread_opt` consistently exhibits superior performance compared to baseline. For baseline, performance scales predictably with increasing threads: doubling the number of threads from 1 to 8 results in approximately halving the runtime. However, the speedup from 8 to 16 threads is 1.67, and from 16 to 32 threads is 1.41, indicating diminishing returns. Notably, for `avx2_multithread_opt` the runtime with 16 threads exceeds that with 8 threads, an anomaly for which we currently have no explanation. Despite this, `avx2_multithread_opt` remains substantially faster than baseline even with 16 threads. `avx2_multithread_opt` achieves a performance improvement factor of 4.06 times over baseline with a single thread and 2.26 times faster with 32 threads. Furthermore, while baseline scales slightly better with additional threads, `avx2_multithread_opt` maintains a clear performance advantage overall.



(a) Baseline CPU utilization over time: The CPU utilization remains consistently high throughout the entire period.



(b) CPU utilization over time for avx2_opt: The CPU utilization is generally around 60%, with occasional spikes.



(c) CPU utilization over time for avx2_multithread_opt: The CPU utilization remains consistently high throughout the observation period, similar to the baseline.

Figure 4.5: CPU utilization comparison for the human genome dataset across different settings: baseline vs avx2_opt vs avx2_multithread_opt

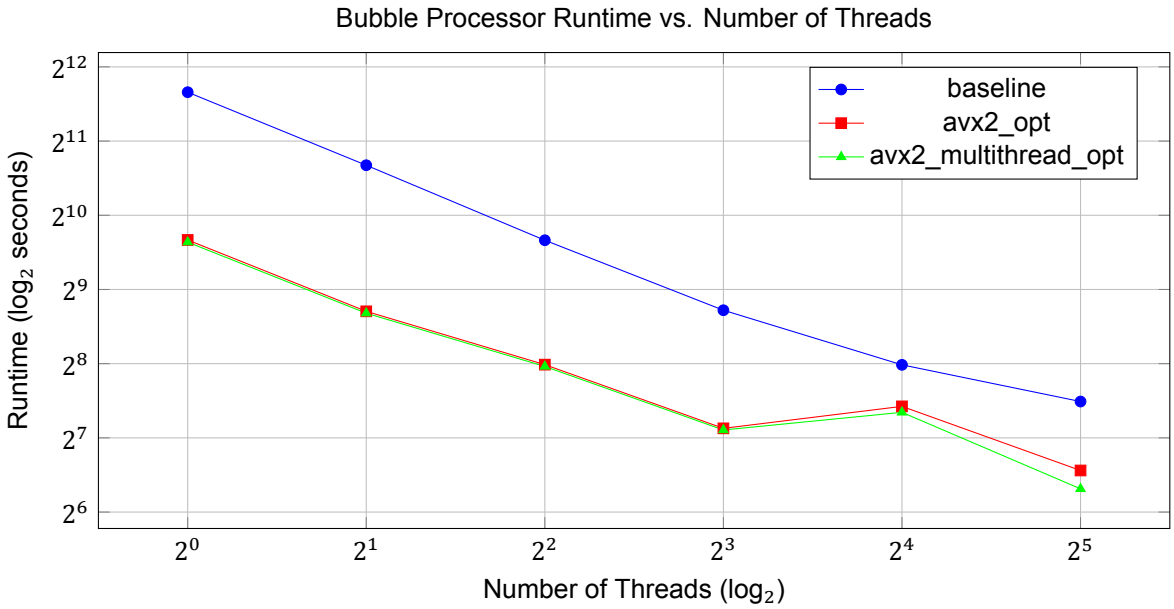


Figure 4.6: Runtime comparison across different threading ($\log_2$ scale for both axes) on 1 million bubbles: baseline vs avx2_multithread_opt

5 Discussion

In this thesis, a novel multithreaded architecture and SIMD instruction set is introduced, evaluating its performance on two distinct datasets: the bacteria dataset and the human genome dataset. The experiments were conducted using two different CPUs: the Intel Xeon Silver 4114 and the AMD EPYC 7532. The results reveal variations in runtime distribution across different datasets and processor configurations. Figure 5.1 builds upon Figure 2.5, with dots representing the speedups achieved for the error correction process under various conditions, as well as the corresponding speedups for the entire polishing stage. Each dot will be discussed in detail in the subsequent sections. For the bacteria dataset, the polishing stage accounts for 61.69% of the total processing time, with the error correction process constituting 69.52% of the polishing stage's duration. When running on a single core, the error correction process achieves speedups of 3.0x and 4.3x with AVX2 and AVX-512 instructions, respectively. This results in overall speedups of 1.9x and 2.1x for the entire polishing process. When utilizing eight cores, the error correction process realizes speedups of 2.6x and 2.7x with AVX2 and AVX-512, leading to overall speedups of 1.7x and 1.8x for the polishing process. In the case of the human genome dataset, the polishing stage represents 20.89% of the processing time, with the error correction process making up 43.47% of this polishing time. Here, the error correction process demonstrates a speedup of 4.0x with AVX2 on a single core when processing 1 million bubbles. With 32 cores, the speedup is 2.3x for the same dataset. The error correction process achieves a 1.4x speedup using AVX2 on 64 cores when applied to the complete dataset, resulting in an overall 1.14x speedup for the entire polishing process. The findings indicate that the proposed multithreaded architecture and SIMD instruction set significantly enhance performance compared to the baseline across both datasets, processors, and thread counts. The improved version demonstrates substantial speedups, confirming its effectiveness in accelerating the polishing process for various data sizes and computational environments.

Several unexpected results emerged from the performance analysis. First, the runtime for the `deletion` function within the `GeneralPolisher` did not decrease as significantly with SIMD instructions compared to other functions. As illustrated in Figure 4.4, the performance improvement for the `deletion` function was less pronounced. In some cases, as shown in Figure 4.6, the runtime even increased compared to the baseline. Switching from the SIMD-optimized version of the `deletion` function to the original implementation in `avx2_opt` and `avx2_multithread_opt` yielded similar runtimes. This similarity can be attributed to the fact that, although the original `deletion` function does not explicitly utilize SIMD instructions, the compiler appears to optimize the code with SIMD instructions at the assembly level. While this compiler optimization explains why the runtime remains relatively constant regardless of whether the SIMD-optimized or original `deletion` function is used for `avx2_opt` and `avx2_multithread_opt`, it does not fully account for the lesser improvement observed with SIMD instructions or, in some instances, the worsened performance.

Second, as shown in Figure 4.3, the performance gains between AVX2 and AVX-512 diminish as the number of threads increases, indicating that additional vectorization optimizations offer diminishing returns at higher levels of parallelism. Specifically, AVX2 and AVX-512 achieve speedups of 3.0x and 4.3x, respectively, on a single core. However, these speedups drop to 2.6x and 2.7x when using eight cores. Similarly, Figure 4.6 demonstrates that the performance improvement from AVX2 over the baseline also decreases with increasing threads. For instance, the error correction process shows a

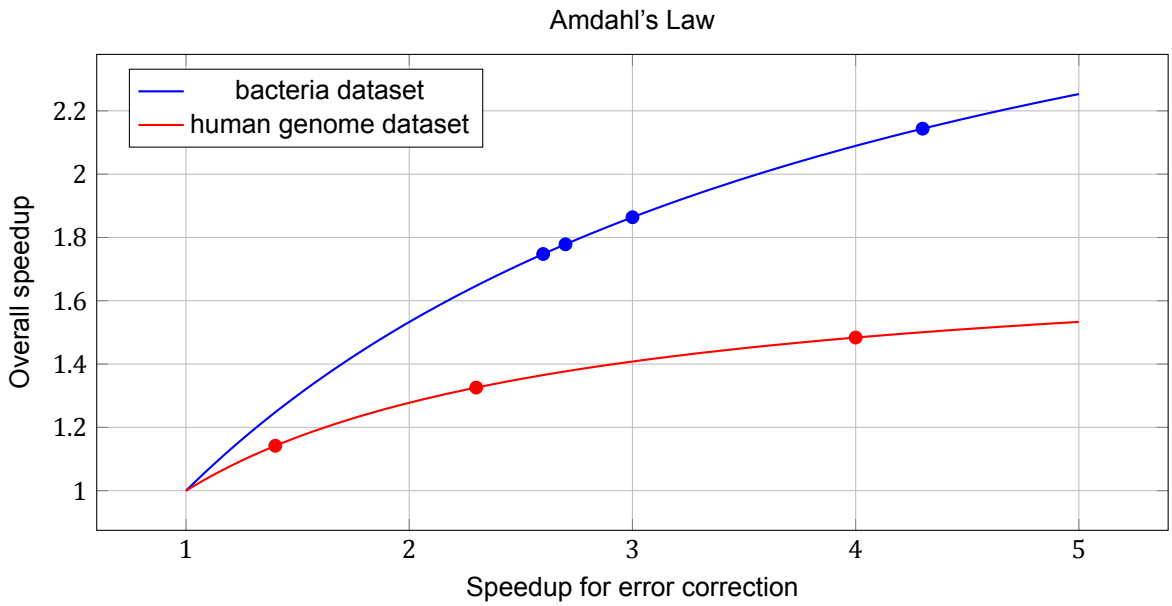


Figure 5.1: Amdahl's Law applied to the Flye polisher, showing overall speedup versus speedup for error correction for two datasets.

4.0x speedup with AVX2 on a single core for processing 1 million bubbles, but this speedup diminishes to 2.3x with 32 cores. One possible reason for this reduction in vectorization benefits at higher core counts can be attributed to other performance bottlenecks, such as memory bandwidth limitations and increased contention for shared resources. Vectorization is most effective when data fits within the cache hierarchy and can be efficiently streamed to vector units. As the number of cores increases, the pressure on memory subsystems and interconnects often offsets the advantages of wider vector registers. Consequently, while vectorization significantly enhances performance on a single core, its impact diminishes as the core count rises if there is contention for shared resources.

6

Conclusions

6.1. Conclusions

In this chapter, we revisit the research questions posed at the beginning of this thesis and evaluate how the findings from the research have addressed them. Each research question will be examined in the context of the solutions and improvements proposed in the study. By doing so, we aim to demonstrate the contributions of this work to the field and outline how the research objectives have been achieved.

Research question 1. How can a new multi-threading architecture be designed to minimize mutex contention and enhance parallel processing in the Flye polishing step? The Flye polisher's error correction process originally used parallel processing to maximize CPU core utilization by reading bubbles from a file into memory and distributing them across multiple threads. Although the polishing process itself was parallelized, both reading and writing operations were handled by a single thread. This created a bottleneck, as writing the polished bubbles to an output file was sequential and caused delays due to mutex contention. The new multithreaded architecture proposed in this thesis addresses these limitations by introducing several key improvements. First, the writing process is parallelized, allowing each thread to write its output to a separate file. These files are then combined at the end of the process. This parallel writing approach reduces the overall writing time and eliminates the need for mutex locks, significantly decreasing wait times. Second, the new design incorporates batch processing. In the original implementation, each thread acquired a mutex lock for every single bubble it processed, resulting in frequent mutex contention. The improved architecture assigns a batch of bubbles to each thread with a single mutex acquisition. This reduces the frequency of mutex locking and unlocking, thereby minimizing contention and enhancing overall performance.

Research question 2. What is the impact of using modern SIMD instruction sets (e.g., AVX2 and AVX-512) on the performance of the Flye polisher? The use of modern SIMD instruction sets, such as AVX2 and AVX-512, significantly impacts the performance of the Flye polisher by enhancing the efficiency of its error correction process. By leveraging advanced vector processing units, multiple data points can be processed simultaneously, accelerating the polishing process for a single bubble on a single core. Each bubble consists of a consensus sequence and numerous segment sequences, with the goal of finding a consensus sequence that best represents all segment sequences. This is achieved by calculating score matrices between the consensus sequence and each segment sequence, followed by iterative mutations to improve the scores. In sequential computing, these score matrices are calculated individually. In contrast, our SIMD implementation employs an inter-read vectorization approach, allowing score matrices to be calculated in batches. The batch size is determined by the specific instruction set used. AVX2, capable of processing 256 bits simultaneously, handles four data points at once, while AVX-512, with a capacity of 512 bits, processes eight data points concurrently. This parallel processing capability of SIMD instruction sets results in a substantial reduction in computation time, thereby improving the overall efficiency of the Flye polisher.

Research question 3. How does the performance of the optimized polisher compare to the original Flye polisher in terms of processing speed and accuracy? The performance comparison between the optimized polisher and the original Flye polisher demonstrates notable differences in processing speed and confirms the preservation of assembly accuracy. Our tests show that the outputs

of both the original and improved versions are identical when run using a single thread. This deterministic approach ensures that the sequence of polished bubbles remains consistent across runs, as using multiple threads can alter this sequence. Once output consistency was verified, we proceeded to evaluate processing speed. The optimized polisher consistently outperforms the original Flye polisher in terms of runtime, with improvements observed under all tested conditions. However, the degree of speedup varies depending on the dataset and the number of threads utilized. Specifically, while the optimized design shows significant speed advantages with fewer threads, the performance gap between the original and optimized polishers narrows as the number of threads increases.

Research question 4. What are the trade-offs between computational speed and resource usage in the optimized Flye polisher? The optimized Flye polisher introduces a trade-off between computational speed and resource usage, particularly in terms of memory. This is evident in the new multi-threaded design, where batch processing is employed. In this design, each thread is assigned a batch of bubbles, which it processes sequentially. Once all the bubbles in a batch are polished, they are collectively written to the output file. This batch writing approach reduces I/O overhead compared to writing each bubble individually. However, it slightly increases memory usage due to the need to store all bubbles in a batch before writing them. For the SIMD implementation, additional memory is consumed due to padding and preprocessing. Padding ensures that all segments in a batch are extended to match the length of the longest segment, which increases the memory required to store score matrices. The space complexity for each score matrix is $O(nm)$, where n is the length of the sequences involved. Thus, padding results in a modest increase in memory usage. Furthermore, the preprocessing step involves computing a 2D matrix to store certain values, with a space complexity of $O(n)$. Although this is relatively minor compared to the score matrix, it still contributes to the overall memory consumption. This increase is especially noticeable when processing large datasets. For example, the baseline approach requires 23 GB of memory, whereas the new approach with a batch size of 10 requires 39 GB, and a batch size of 100k requires 209 GB. This increase is minimal compared to the 507GB memory requirement of the previous stage, underscoring the efficiency of the optimized design in balancing computational speed with resource usage.

In summary, the new multithreaded architecture implemented in the design increases the portion of the process that can be parallelized, minimizes the time required for acquiring mutex locks and reduces waiting time. Additionally, SIMD instructions were utilized for the polishing process, leveraging the vector processor unit's capability to process 4 or 8 data points simultaneously. This approach accelerated the polishing process. As a result, the output of the improved version remains identical to previous versions, while the total runtime decreases across various settings (different datasets, different thread counts) due to the multithreaded architecture, which reduces waiting time, and the SIMD instructions, which reduce processing time. Although the new version increases memory usage, the peak RAM consumption remains significantly lower than the peak RAM required for other stages in the Flye workflow.

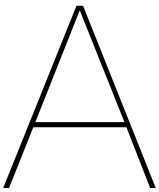
6.2. Recommendations

In this thesis, we introduced significant improvements to the Flye polisher, enhancing its performance through parallelized multithreading and SIMD instruction sets. While these optimizations have demonstrated gains in efficiency and speed, there remains potential for further development.

In future work, it is important to investigate the diminishing performance gains observed with increasing parallelism when using vectorization optimizations such as AVX2 and AVX-512. The data shows that while these optimizations provide substantial speedups on a single core, their effectiveness decreases as the number of threads increases. This reduction in vectorization benefits at higher core counts is likely due to bottlenecks such as memory bandwidth limitations and increased contention for shared resources. Future research could explore strategies to mitigate these bottlenecks and enhance the scalability of vectorized code in multi-core environments.

Future work can also focus on leveraging advancements in GPU technology to further accelerate the Flye polisher. GPUs have emerged as a powerful tool for accelerating high-performance computing tasks due to their ability to handle large-scale parallel processing. Unlike traditional CPUs, which have a limited number of cores optimized for sequential processing, GPUs contain thousands of smaller cores designed to execute many threads simultaneously. This architecture makes GPUs particularly well-suited for data-parallel tasks that can be broken down into smaller, independent computations.

Given the parallel nature of the Flye polisher’s error correction process, integrating GPU acceleration presents an exciting opportunity for future research.



Appendix

Type	Event	Probability
mat	A	0.958
mat	C	0.944
mat	G	0.950
mat	T	0.956
mis	A->C	0.005
mis	A->G	0.002
mis	A->T	0.002
mis	C->A	0.008
mis	C->G	0.004
mis	C->T	0.004
mis	G->A	0.004
mis	G->C	0.003
mis	G->T	0.004
mis	T->A	0.004
mis	T->C	0.003
mis	T->G	0.004
del	A	0.032
del	C	0.041
del	G	0.039
del	T	0.033
ins	A	0.027
ins	C	0.019
ins	G	0.022
ins	T	0.021
noins		0.912

Table A.1: Statistical Parameters of the P6-C4 Protocol

Algorithm 3 deletion

Require: Index $letterIndex$, reads $reads$; Forward and reverse score matrices $forwardScores$, $reverseScores$
Ensure: Final score
Initialize $finalScore \leftarrow 0$
$frontRow \leftarrow letterIndex - 1$
$revRow \leftarrow letterIndex$
for $readId \leftarrow 0$ to $|reads|$ **do**
  $forwardScore \leftarrow forwardScores[readId]$
  $reverseScore \leftarrow reverseScores[readId]$
  $maxVal \leftarrow$ lowest possible value
  $cols \leftarrow |reads[readId]|$
  for $col \leftarrow 0$ to $cols$ **do**
    $sum \leftarrow forwardScore[frontRow, col] + reverseScore[revRow, col]$
    $maxVal \leftarrow \max(maxVal, sum)$
  end for
  $finalScore \leftarrow finalScore + maxVal$
end for
return $finalScore$

Algorithm 4 substitution

Require: Index $letterIndex$, character $base$, reads $reads$; Forward and reverse score matrices $forwardScores$, $reverseScores$
Ensure: Final score
Initialize $finalScore \leftarrow 0$
$frontRow \leftarrow letterIndex - 1$
$revRow \leftarrow letterIndex$
$baseScoreWithGap \leftarrow getScore(base, '-')$
for $readId \leftarrow 0$ to $|reads|$ **do**
  $forwardScore \leftarrow forwardScores[readId]$
  $reverseScore \leftarrow reverseScores[readId]$
  $cols \leftarrow |reads[readId]|$
  $maxVal \leftarrow forwardScore[frontRow, 0] + reverseScore[revRow, 0] + baseScoreWithGap$
  for $col \leftarrow 0$ to $cols - 1$ **do**
    $readBase \leftarrow reads[readId][col]$
    $match \leftarrow forwardScore[frontRow, col] + getScore(base, readBase)$
    $ins \leftarrow forwardScore[frontRow, col + 1] + baseScoreWithGap$
    $maxVal \leftarrow \max(maxVal, \max(match, ins) + reverseScore[revRow, col + 1])$
  end for
  $finalScore \leftarrow finalScore + maxVal$
end for
return $finalScore$

Algorithm 5 insertion

Require: Position pos, character $base$, reads $reads$; Forward and reverse score matrices $forwardScores, reverseScores$

Ensure: Final score

Initialize $finalScore \leftarrow 0$
$frontRow \leftarrow pos - 1$
$revRow \leftarrow pos - 1$
$baseScoreWithGap \leftarrow getScore(base, '-')$
for $readId \leftarrow 0$ to $|reads|$ **do**
  $forwardScore \leftarrow forwardScores[readId]$
  $reverseScore \leftarrow reverseScores[readId]$
  $cols \leftarrow |reads[readId]|$
  $maxVal \leftarrow forwardScore[frontRow, 0] + reverseScore[revRow, 0] + baseScoreWithGap$
  for $col \leftarrow 0$ to $cols - 1$ **do**
    $readBase \leftarrow reads[readId][col]$
    $match \leftarrow forwardScore[frontRow, col] + getScore(base, readBase)$
    $ins \leftarrow forwardScore[frontRow, col + 1] + baseScoreWithGap$
    $maxVal \leftarrow \max(maxVal, \max(match, ins) + reverseScore[revRow, col + 1])$
  end for
  $finalScore \leftarrow finalScore + maxVal$
end for
return $finalScore$

Bibliography

[1] Tanveer Ahmad et al. “Optimizing performance of GATK workflows using Apache Arrow In-Memory data framework”. In: *BMC genomics* 21 (2020), pp. 1–14.

[2] Nauman Ahmed et al. “Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm”. In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2015, pp. 240–246.

[3] 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 (2012), pp. 1–18.

[4] David Deamer, Mark Akeson, and Daniel Branton. “Three decades of nanopore sequencing”. In: *Nature biotechnology* 34.5 (2016), pp. 518–524.

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

[6] Elena Espinosa et al. “Advancements in long-read genome sequencing technologies and algorithms”. In: *Genomics* (2024), p. 110842.

[7] Borja Freire, Susana Ladra, and José R Paramá. “Memory-efficient assembly using Flye”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.6 (2021), pp. 3564–3577.

[8] Ernst Joachim Houtgast et al. “Hardware acceleration of BWA-MEM genomic short read mapping for longer read lengths”. In: *Computational biology and chemistry* 75 (2018), pp. 54–64.

[9] Saurabh Kalikar et al. “Accelerating minimap2 for long-read sequencing applications on modern CPUs”. In: *Nature Computational Science* 2.2 (2022), pp. 78–83.

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

[11] 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.

[12] Heng Li. *Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM*. 2013. arXiv: 1303.3997 [q-bio.GN]. URL: https://arxiv.org/abs/1303.3997.

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

[14] 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.

[15] Glennis A Logsdon, Mitchell R Vollger, and Evan E Eichler. “Long-read human genome sequencing and its applications”. In: *Nature Reviews Genetics* 21.10 (2020), pp. 597–614.

[16] QX Charles Mak et al. “Polishing de novo nanopore assemblies of bacteria and eukaryotes with FMLRC2”. In: *Molecular Biology and Evolution* 40.3 (2023), msad048.

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

[18] Shanshan Ren et al. “GPU accelerated sequence alignment with traceback for GATK HaplotypeCaller”. In: *BMC genomics* 20 (2019), pp. 103–116.

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

[20] Temple F Smith, Michael S Waterman, et al. “Identification of common molecular subsequences”. In: *Journal of molecular biology* 147.1 (1981), pp. 195–197.

[21] Robert Vaser et al. “Fast and accurate de novo genome assembly from long uncorrected reads”. In: *Genome research* 27.5 (2017), pp. 737–746.

[22] Md Vasimuddin et al. “Efficient architecture-aware acceleration of BWA-MEM for multicore systems”. In: *2019 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE. 2019, pp. 314–324.

[23] Justin M Zook et al. “Extensive sequencing of seven human genomes to characterize benchmark reference materials”. In: *Scientific data* 3.1 (2016), pp. 1–26.