Performance-Driven Algorithm Engineering

Optimising Pairwise Sequence Alignment and Pattern Matching Algorithms in the Era of Pangenomic Sequence Analysis

Dissertation

zur Erlangung des Grades eines Doktors
der Naturwissenschaften (*Dr. rer. nat.*)

am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

vorgelegt von

René Rahn

Berlin 2023

Erstgutachter: **Prof. Dr. Knut Reinert**

Zweitgutachter: **Prof. Dr. Bernhard Renard**

Tag der Disputation: 15.12.2023

für Yasmin, Kilian und Thea

Abstract

A number of technological advancements in high-throughput genome sequencing have led to the generation of exabyte-scale sequencing data worldwide in recent years. These developments have facilitated large-scale resequencing projects like the 1000 Genomes Project, which aim to catalog the genetic diversity of organisms and specific populations. In the context of medical research, incorporating population data, is of particular interest. However, existing applications are often optimised for analysing a few sequences, and the methods used cannot be easily transferred to these vast datasets without exceeding system resources or producing results within a suitable time frame.

Simultaneously, the execution model of processors has evolved from sequential to highly parallel process execution, thanks to the addition of processor cores, vector processing units, and advances in superscalar processor designs. This ongoing development in high-performance computing requires applications and algorithms to scale with the growing levels of parallelism. However, highly optimised algorithms are often embedded within applications, making them practically inaccessible to the scientific community.

In this dissertation, we first investigate a generic approach to parallelise and vectorise pairwise sequence alignments using a dynamically scalable concurrency model. We explore various techniques and code-level optimisations to effectively utilise the available parallelisms on modern high-performance CPUs. The results demonstrate that the dynamically accelerated pairwise sequence alignment scales proportionally with the number of cores and provides speed-ups of up to a factor of 2500 compared to the sequential reference implementation on modern hardware.

Second, we propose a general solution for data-compressed acceleration of pattern matching algorithms by compressing a large collection of related DNA sequences and providing a set of composable algorithms to refine and optimise the applicable operations. Our research on data-compressed acceleration shows hundred-fold speed-ups for online searching over a pangenome comprising over 5000 reference sequences compared to the naive approach of individually searching all sequences. These speed-ups are achieved while utilising only a fraction of the main memory.

Moreover, we implemented these features in dedicated modules of the open-source software library SeqAn (`https://www.seqan.de/`), making them accessible and adoptable by the entire research community. While doing so, we strived for a user-friendly API design so that these methods can be easily customised and extended, making them applicable to a wide range of applications in the domain of computational sequence analysis.

Zusammenfassung

Eine Reihe von technologischen Fortschritten bei der Hochdurchsatz-Sequenzierung von Genomen hat in den letzten Jahren weltweit zur Erzeugung von Sequenzierungsdaten im Exabyte-Bereich geführt. Diese Entwicklungen haben groß angelegte Resequenzierungsprojekte, wie das 1000 Genomes Project, ermöglicht, welche darauf abzielen die genetische Vielfalt ganzer Populationen von unterschiedlichen Organismen zu katalogisieren. Bestehende Anwendungen sind jedoch häufig für die Analyse einiger weniger Sequenzen optimiert. Häufig lassen sich die verwendeten Methoden nicht ohne weiteres auf solche riesigen Datenmengen übertragen, ohne dabei die zur Verfügung stehenden Rechenressourcen zu übersteigen oder in angemessener Zeit entsprechende Ergebnisse zu produzieren.

Gleichzeitig führten kontinuierliche Verbesserungen in der Prozessorherstellung zu einem fundamentalen Wechsel im Programmiermodell von einer sequentiellen hin zu einer hochparallelen Ausführung von Prozessen. Diese fortlaufende Entwicklung im Bereich des Hochleistungsrechnens erfordert, dass Anwendungen und Algorithmen mit der zunehmenden Parallelität skalieren. Optimierte Algorithmen sind jedoch häufig in konkrete Anwendungen eingebettet, so dass sie für die wissenschaftliche Gemeinschaft praktisch unzugänglich sind.

In dieser Dissertation untersuchen wir zunächst einen generischen Ansatz zur Parallelisierung und Vektorisierung paarweiser Sequenzalignments unter Verwendung eines skalierbaren Nebenläufigkeitsmodells. Dabei betrachten wir verschiedene Optimierungsansätze, um die verfügbaren Parallelitäten moderner Hochleistungs-CPUs effektiv zu nutzen. Unsere Ergebnisse zeigen, dass wir im Vergleich zur sequenziellen Ausführung auf modernen CPUs Geschwindigkeitssteigerungen von bis zu einem Faktor von 2500 erreichen.

Im zweiten Teil betrachten wir Methoden zur datenkomprimierten Beschleunigung von Pattern-Matching-Algorithmen. Dabei komprimieren wir eine große Sammlung ähnlicher DNA-Sequenzen anhand einer Referenzsequenz und entwickeln eine Reihe von kombinierbaren Algorithmen, mit deren Hilfe existierende Suchalgorithmen auf dem datenkomprimierten Format angewendet werden können. Durch die Verarbeitung in komprimierter Form, konnten wir, verglichen mit dem naiven Ansatz, hundertfache Geschwindigkeitssteigerungen bei der Online-Suche in einem Pangenom mit über 5000 Referenzsequenzen erzielen.

Darüber hinaus haben wir die entwickelten Funktionen und Operationen in konkreten Modulen der Open-Source-Softwarebibliothek SeqAn (https://www.seqan.de/) implementiert, so dass sie für die gesamte Forschungsgemeinschaft zugänglich sind. Dabei haben wir uns um ein benutzerfreundliches API-Design bemüht, so dass unsere Methoden leicht adaptiert und erweitert werden können und für ein breites Spektrum von Anwendungen im Bereich der computergestützten Sequenzanalyse nutzbar sind.

Acknowledgments

I would like to express my deepest gratitude and appreciation to all those who have contributed to the completion of this doctoral thesis.

First and foremost, I am immensely grateful to my advisor, Prof. Dr. Knut Reinert, for his unwavering guidance, invaluable insights, and constant support throughout this research journey. His expertise, patience, and encouragement have been instrumental in shaping this work.

I would like to extend my gratitude to Edzard Höfig and Martin Riese, who generously contributed their time to proof read and appraise this thesis.

Special thanks go to David Weese who supervised me in the beginning of my research and gave me constant guidance while working together on the journaled sequences and to the Federal Ministry of Education and Research for their financial support.

I further would like to extend my appreciation to Jonny Hancox, Marcel Ehrhardt, Pascal Costanza, and Stefan Budach, for our collaborative work on the acceleration of pairwise sequence alignments. I am also grateful to Denis Bakhvalov for giving me invaluable insights into performance optimisations, while contributing a vectorisation lab to his ninja-perf online course.

To Matthias, Florian, and Thomas from ZIB, as well as Klaus-Dieter, Heinrich, and Thorsten from Intel for their technical assistance and support during our joint time at the IPCC project.

To Alexander, Daniel, Jenny, Julianus, and Temesgen, for their knowledge and collaboration in organising and holding numerous training sessions and to Leon and Sabrina for delightful evenings in Heidelberg during my time at the de.NBI project.

Many thanks to the current and former members of the Bioinformatics and Software Engineering groups at the Freie Universität Berlin and all other colleagues from the RKI, MPI, KNIME, Intel, Eberhard Karls University Tübingen, and de.NBI, I had the pleasure to working with. Our collaborative discussions and brainstorming sessions, as well as the great atmosphere during and after work and on various workshops have significantly contributed to this research, which would not have been possible without your cooperation and participation. Thank you for being an integral part of this experience.

To my family and friends, I am deeply grateful for your unwavering support, encouragement, and understanding throughout this journey. I dedicate this work to Yasmin, Kilian, and Thea, for your unconditional love, patience, and belief in me, which has been my driving force, providing the motivation and strength to overcome all challenges, persevere, and bringing this work to fruition.

Contents

1 Introduction **1**
- 1.1 Promises and challenges of personalised genomics . . . 1
- 1.2 Research objectives . . . 4
- 1.3 Outline . . . 5

2 Preliminaries **7**
- 2.1 Fundamental terms and definitions . . . 7
- 2.2 Sequences . . . 9
- 2.3 Sequence alignment . . . 11
  - 2.3.1 Alignment representation . . . 11
  - 2.3.2 Alignment score . . . 13
  - 2.3.3 Optimal pairwise alignment . . . 16
  - 2.3.4 Dynamic programming . . . 17
- 2.4 Pattern matching . . . 23
  - 2.4.1 Online methods . . . 23
  - 2.4.2 Filter methods . . . 25

3 Hardware-accelerated pairwise alignment **31**
- 3.1 Background . . . 31
  - 3.1.1 Terminologies and peformance metrics . . . 33
  - 3.1.2 Memory hierarchy . . . 34
  - 3.1.3 Data-level parallelism . . . 36
  - 3.1.4 Thread-level parallelism . . . 40
  - 3.1.5 Instruction-level parallelism . . . 42
- 3.2 Related work . . . 44
  - 3.2.1 Inter-sequence execution . . . 46
  - 3.2.2 Intra-sequence execution . . . 47
  - 3.2.3 Multi-level parallelism . . . 47
- 3.3 Utilising data-level parallelism . . . 48
  - 3.3.1 Data representation . . . 48
  - 3.3.2 Memory transformation . . . 50
  - 3.3.3 Vectorised scoring schemes . . . 52
  - 3.3.4 Processing heterogeneous sequence collections . . . 53
  - 3.3.5 Maximising SIMD throughput . . . 59
- 3.4 Utilising thread-level parallelism . . . 65
  - 3.4.1 Dynamic scheduling and execution . . . 65
  - 3.4.2 Intra-sequence execution . . . 68
- 3.5 Improving cache efficiency and instruction-level parallelism . . . 71
  - 3.5.1 Striped execution pattern . . . 71

3.5.2 Loop unrolling . . . 72
3.6 Applications and evaluation . . . 72
3.6.1 Test systems . . . 73
3.6.2 Evaluation of short bulk alignment requests . . . 73
3.6.3 Evaluation of long single alignment requests . . . 76
3.6.4 Evaluation of long bulk alignment requests . . . 79
3.6.5 Evaluation of novel improvements . . . 82
3.7 Universal algorithm design . . . 86

4 Compression-accelerated pattern matching **91**
4.1 Background . . . 91
4.1.1 DNA sequencing . . . 92
4.1.2 Resequencing . . . 92
4.1.3 Reference bias . . . 93
4.1.4 Computational pangenomics . . . 93
4.2 Related work . . . 95
4.2.1 Graph-based pangenome structures . . . 96
4.2.2 Reference-centric pangenome models . . . 97
4.3 Referentially compressed multisequence . . . 100
4.3.1 Data representation . . . 100
4.3.2 Implementation details . . . 104
4.3.3 Construction . . . 108
4.3.4 Data retrieval . . . 112
4.4 Dynamic traversal interface . . . 124
4.4.1 Journaled sequence tree . . . 125
4.4.2 Tree polishing . . . 130
4.4.3 Universal traversal . . . 133
4.4.4 Tree transformation and tracing . . . 136
4.5 Application and evaluation . . . 142
4.5.1 Benchmark data . . . 142
4.5.2 Online pattern matching . . . 144
4.5.3 Read mapping . . . 147

5 Summary and future work **153**

Bibliography **171**

List of Figures **173**

List of Tables **175**

List of Algorithms **177**

Index **179**

Abbreviations **183**

1 Introduction

> The completion of the Human Genome Project in 2001 opened the floodgates to a deeper understanding of medicine.
>
> *(Carrasco-Ramiro et al.)*

Since the publication of the first human draft sequence at the beginning of the 21st century [Lander et al., 2001; Venter et al., 2001], the role of *genomics* in healthcare has undergone a dramatic transformation. This can be attributed to two key factors. Firstly, significant advancements have been made in DNA (deoxyribonucleic acid) sequencing technology, commonly referred to as *next-generation sequencing* (NGS). These innovative technologies have facilitated the acquisition of large amounts of genetic sequences through massively parallel sequencing reactions. Consequently, the sequencing throughput has been substantially increased while reducing costs compared to the previous dominant Sanger sequencing technology [Goodwin et al., 2016; Metzker, 2010; Shendure and Ji, 2008; Slatko et al., 2018]. Secondly, powerful bioinformatic analysis routines have been developed to extract meaningful insights from the acquired sequencing data and identify various genetic alterations [Bernhardsson et al., 2020; Di Resta et al., 2018; McGuire et al., 2020; Oliva et al., 2021; Pabinger et al., 2014; Pereira et al., 2020; Wang et al., 2013; Wee et al., 2019].

1.1 Promises and challenges of personalised genomics

Particularly, in cancer patient care and in diagnosing rare diseases, of which 80% are known to be related to a genetic condition [Smedley et al., 2021], one can observe a shift from standard molecular testing towards diagnostic procedures aided by NGS analysis [Qin, 2019; Ramoni et al., 2017; Smedley et al., 2021; Turro et al., 2020], also known as *precision medicine* [Cardon and Harris, 2016; Carrasco-Ramiro et al., 2017; Personalized Medicine Coalition, 2021; The Centre for Personalised Medicine, 2021]. The vision and central objectives of these new medical approaches are to utilise NGS technologies to design and administer targeted therapies based on the individual genetic composition of the patients. Among others, this includes the development of new drugs with improved pharmacokinetics [Harper and Topol, 2012; Relling and Evans, 2015; Spreafico et al., 2020], and enhanced risk management of genetic diseases [Gurdasani et al., 2019; Khera et al., 2018; Wei et al., 2009; Weitzel et al., 2011], '...to ensure that patients get the right treatment at the right dose and at the right time.' [Gen, 2016].

The following case study about a four-year-old girl who suffered from a rare disease known as the GLUT1-deficiency syndrome [Gen, 2016; Cookson, 2017] illustrates this representatively for diagnosing rare diseases. An aberration in her SLC2A1 gene led to an inhibition of the glucose transponder protein type 1 (GLUT1), which is needed for glucose uptake into the brain [Leen et al., 2010; Soto-Insuga et al., 2019; Wang et al., 2005], manifesting in

severe neurological symptoms such as epileptic seizures, slow mental development, and coordination issues [Gen, 2016]. Because her diffuse symptoms could be explained by different neurological disorders, physicians could not find the root cause of her symptoms through standard molecular tests, and she had to undergo numerous clinical visits [Cookson, 2017]. It was only after enlisting in the 100 000 *Genomes Project* (100KGP) [Genomics England et al., 2020], where researchers compared her DNA with that of her parents, that her symptoms could be linked to a mutation of the SLC2A1 gene. Eventually, her condition could be improved by successfully administering a ketogenic diet [Cookson, 2017; Leen et al., 2010].

Population-scale sequencing

A large proportion of the population experiences a similar situation to this girl. In fact, a review by Ferreira [2019] revealed that there are approximately 10 000 rare diseases affecting 6% of people in Western society. Likewise, cancer puts a high burden on society, as millions of people are affected every year. In 2020, cancer surveillance detected 19.3 million new cases of cancer worldwide [Ferlay et al., 2021]. Since both diseases are caused by genetic aberrations, they are commonly associated with a high mortality rate, many years of life lost, and a significantly lowered quality of life, while patients need to endure countless tests and require intensive medical care.

Consequently, there is increased interest in developing fast and precise medical diagnostic routines, as well as effective therapies with minimal adverse effects. This has led to the launch of multiple population-scale sequencing endeavours like the 1000 *Genomes Project* (1KGP) or the aforementioned 100KGP in the last two decades. Such sequencing projects pursue the ambitious goal of cataloguing the genetic diversity of entire populations or specific cohorts associated with a particular disease [Carrasco-Ramiro et al., 2017]. This involves the generation of variant databases covering sequences from a few thousand individuals, such as the 1KGP launched in 2005 [Auton et al., 2015], up to a million individuals, as seen in the Precision Medicine Initiative announced 10 years later [Collins and Varmous, 2015].

Data growth

The extensive collection of genetic data from population-scale sequencing projects, however, presents significant challenges in handling and analysing such vast amounts of information, thereby giving rise to the emerging field of big data in genomics. In fact, Schork [2015] estimated the growth rate to double between every seven to every twelve months (see Fig. 1.1). Based on these growth rates, the authors projected an annual sequencing capacity of 1 ZB (zettabyte) – the equivalent of 1 billion TBs (terabyte)– by 2025, requiring storage capacities of 2 to 40 EBs (exabyte) for human genome sequencing alone. For example, the *Sequencing Read Archive* (SRA) [Leinonen et al., 2011] – a database collecting NGS data from human, non-human, and microbial sources – contained more than 36 PB (petabyte) in 2019 and is projected to grow to 43 petabytes by 2023 [National Institutes of Health, 2020]. Evidently, these enormous volumes of data not only pose a challenge for data centres to host sequencing data in a sustainable manner but also require new strategies for efficient distribution and decentralised use of the available data in different experimental settings

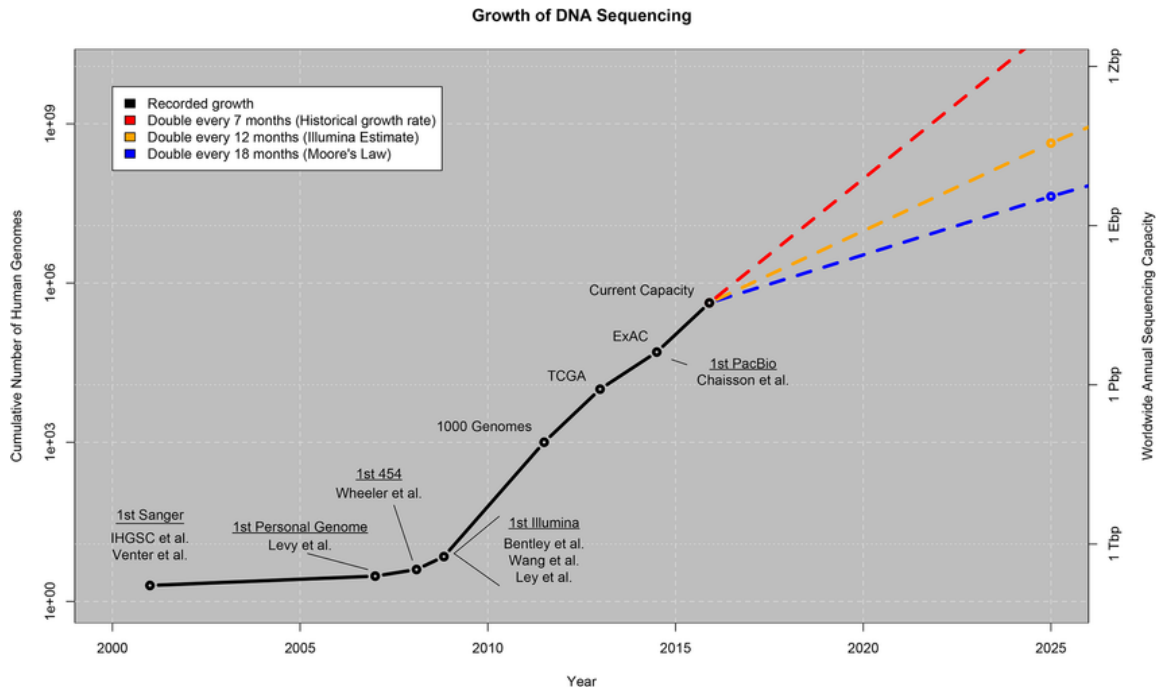


Figure 1.1: The plot shows the growth of DNA sequencing both in the total number of human genomes sequenced (left axis) as well as the worldwide annual sequencing capacity (right axis: Tbp (tera-basepairs), Pbp (peta-basepairs), Ebp (exa-basepairs), Zbp (zetta-basepairs)). The values through 2015 are based on the historical publication record, with selected milestones in sequencing (first Sanger through first PacBio human genome published) as well as three exemplar projects using large-scale sequencing: the 1KGP, aggregating hundreds of human genomes by 2012 [Auton et al., 2015]; *The Cancer Genome Atlas* (TCGA), aggregating over several thousand tumor/normal genome pairs [Chin et al., 2011]; and the *Exome Aggregation Consortium* (ExAC), aggregating over 60 000 human exomes [Lek et al., 2016]. Many of the genomes sequenced to date have been whole exome rather than whole genome, but we expect the ratio to be increasingly favoured towards whole genome in the future. The values beyond 2015 represent our projection under three possible growth curves. Plot and caption taken from [Schork, 2015]

that typically vary widely in terms of available computational and financial resources [Muir et al., 2016].

Leveraging computing power of modern CPUs

To take advantage of these developments, it is necessary that analyses also scale with the vast amount of acquired sequencing data. This requires, in addition to developing improved algorithms and data structures, optimal utilisation of the available computing power of modern CPUs (central processing units) and other accelerators, e.g. *graphics processing units* (GPUs) and *field-programmable gate arrays* (FPGAs). CPU performance, alone, has significantly improved over the last 20 years due to sophisticated processor designs. For example the Intel Xeon Scalable processor family supports high-end server configurations, where a single 2-socket CPU with 56 cores can reach a theoretical peak performance of over 4 TFLOP/s [Vladimirov, 2017], which comes primarily from increased parallelism and

enhanced memory systems. To leverage such a peak performance, however, the software must be able to scale with the compute resources added to a single multicore CPU by shifting from a sequential to concurrent execution. This, in turn, leads to greater complexity in software design since transforming the algorithms used from serial execution to concurrent execution is often not trivial, tedious, and error-prone [Leiserson et al., 2020]. Moreover, it requires a deep understanding of the algorithms, the targeted processor architectures, and how processes are executed in the hardware.

The need to address the challenges arising from the vast amount of acquired sequencing data and the increasing complexity of software design in genomic analyses is driven by these observations. This entails designing and developing improved algorithms and data structures that effectively leverage the different levels of parallelism provided by modern CPUs. These algorithms and data structures should also have the capability to scale with the substantial amount of acquired sequencing data. Additionally, it is crucial to enhance the accessibility and usability of the software, ensuring that it is widely available and usable within the entire research community.

1.2 Research objectives

The focus of this thesis lied on modernising and optimising two crucial classes of algorithms extensively used in computational sequence analysis: *pairwise sequence alignments* and *pattern matching algorithms*. To accelerate the computation of pairwise sequence alignments on modern CPUs, we developed various code-level optimisations. We also engineered a succinct data representation based on differential encoding to efficiently manage large collections of similar sequences. We leveraged this compressed representation to accelerate existing online pattern matching algorithms that normally operate on linear sequences. Note we limit our work to modern CPU architectures, as covering additional high-performance accelerators like GPUs or FPGAs would exceed the scope of this thesis. Nonetheless, there are many overlaps in the general software design that applies independent of the programming model to different processor architectures.

We integrated all implementations as reusable modules into the C++ software library *SeqAn* [Döring et al., 2008; Reinert et al., 2017]. The SeqAn project, initiated in 2008 at Freie Universität under the guidance of Prof. Dr. Knut Reinert, has long served as a valuable programming resource for the bioinformatics community, with a strong emphasis on efficient sequence analysis. Given the broad application scope of this library, a key aim of this work was to establish generic interfaces that can be utilised in wide range of scenarios, providing future developers with a well-tested, user-friendly, and extensively documented infrastructure. The concrete objectives of this work can be summarised as follows:

- Conceptualising and developing a unified design for the various pairwise sequence alignment algorithms implemented in SeqAn, with a focus on enhancing their maintainability and extensibility.
- Investigating and implementing appropriate concurrency models to achieve efficient and scalable acceleration of pairwise sequence alignments, considering different levels of parallelism provided by modern CPUs.

- Developing a compact data representation that can effectively scale with population-scale sequencing data.
- Extending this data representation to provide a compression-accelerated interface, allowing seamless integration of existing online algorithms.

1.3 Outline

The thesis is structured into two main chapters, each addressing specific aspects of the research objectives. Before we explain the methodologies in detail, we provide the necessary foundational knowledge in the next chapter Chapter 2. It covers fundamental terms and definitions related to genomics. The chapter briefly summaries important aspects of sequence alignments, including alignment representation, alignment scores, and the algorithmic foundation of computing optimal pairwise alignments with dynamic programming. Subsequently, it explores strategies for pattern matching, including exact and approximate matching, and briefly recalls well-known online methods as well as filter-based methods in this area. Readers familiar with these topics may as well skip this chapter and continue with one of the main chapters Chapter 3 or Chapter 4. Note these chapters do not depend on each other and can be read in any order.

Chapter 3 focuses on hardware-accelerated computation of pairwise alignments and covers our work published in:

Rahn, R. et al. Generic accelerated sequence alignment in SeqAn using vectorization and multi-threading. *Bioinformatics*, 34(20):3437–3445, oct 2018. ISSN 1367-4803. doi: 10.1093/bioinformatics/bty380. URL `https://academic.oup.com/bioinformatics/article/34/20/3437/4992147`

It begins with a background section that gives a brief overview of the memory hierarchy and different levels of parallelism supported on modern multicore CPUs and discusses the potential and challenges of leveraging these factors followed by a review of related work to accelerate the computation of pairwise sequence alignments using multithreading and vectorisation. The chapter then dives into the detailed aspects of our approach to utilise data-level parallelism in order to accelerate the computation of pairwise sequence alignments. It covers the topics of efficient data representation, memory transformation, and vectorised scoring schemes and includes novel methods to maximise the data-level parallelism. Subsequently, the chapter describes our methods to combine the vectorisation approaches with the thread-level parallelism via dynamic scheduling and execution techniques. The chapter concludes with the evaluation of various applications and analysis of the achieved performance gains as well as an overview of the underlying software and API design.

Chapter 4 addresses the topic of compression-accelerated pattern matching and builds on our initial work published in:

Rahn, R., Weese, D. and Reinert, K. Journaled string tree-a scalable data structure for analyzing thousands of similar genomes on your laptop. *Bioinformatics*, 30(24):3499–3505, 2014. ISSN 14602059. doi: 10.1093/bioinformatics/btu438

This chapter briefly introduces potential and challenges of computational pangenomics, before it reviews related work using graph-based pangenome structures and reference-centric

pangenome models. After this, it gives a formal description of our data representation to efficiently model multiple sequences using referential sequence compression including essential operations to operate on the compressed sequence data. It thereby covers methodologies to dynamically search the compressed sequences with any online algorithm using a tree traversal approach. The chapter concludes with the evaluation of the performance of different pattern matching algorithms and provides a design of a read mapping application that can be applied to a reference panel consisting of thousands of sequences.

Finally, Chapter 5 summarises the main contributions made in this thesis and discusses potential directions for future research in the field.

2 Preliminaries

This chapter serves as an introduction to the key terminology and definitions that are essential for understanding the methodologies discussed in this thesis. We start by recalling fundamental mathematical concepts of set theory and formal language theory. Subsequently, we provide a concise overview of the algorithmic principles underlying pairwise alignment and pattern matching algorithms. Please note that this chapter is a refresher only and does not aspire to discuss the topics presented in detail or to be complete.

2.1 Fundamental terms and definitions

We define a *value* as an indivisible entity that remains constant. For instance, the number 10 can be considered a value. Values can be assigned to *objects*, which are also referred to as *variables*. In the context of computers, an object represents a memory address where the assigned value can be stored. To assign the value 10 to an object a, we use the notation $a \leftarrow 10$. Additionally, the expression $a = 10$ signifies that the object a contains the value 10. The type of an object determines the range of values that can be stored within it. Mathematically, this can be represented in form of a *set*, which is a collection of unique values. We may also write *elements* of a set. Table 2.1 provides a compilation of common

Notation	Description
$\{1, 2, 3, 4\}$	A set over the values $1, 2, 3$ and 4
$\{1, 3, 5 \dots\}$	A set over all positive odd integers, where $\dots$ represents a continuation of the pattern
$\{a \mid 0 < a\}$	A set over all positive numbers expressed in set builder notation
$\lvert\{1, 2, 3, 4\}\rvert = 4$	The cardinality of a set, denoting the number of values contained in the set; we may also use *length* or *size*
$a \in \{1, 2, 3, 4\}$	Set membership to specify that the object a corresponds to one of the four values $1, 2, 3$, or 4

Table 2.1: Common set notations.

set notations and Table 2.2 of fundamental sets that we use throughout this thesis. Note that $[j..i) = \emptyset$ also denotes the empty set.

Definition 2.1.1 (Relation)**.** Given two sets A and B, then a *relation* R from A to B is a subset of the crossproduct of both sets, i.e. $R \subseteq A \times B$. Moreover, a subset of only the first elements of the ordered pairs defined by the relation R is called the *domain* of R and a subset of only the second elements of those is called the *range* of R. Specifically, we write

Notation	Description
$\emptyset$	The empty set
$\mathbb{N}$	The set of all positive integers, i.e. $\{1, 2, 3, \dots\}$
$\mathbb{N}_0$	The set of all non-negative integers, i.e. $\{0, 1, 2, 3, \dots\}$
$\mathbb{Z}$	The set of all integers, i.e. $\{\dots, -2, -1, 0, 1, 2, \dots\}$
$\mathbb{R}$	The set of all real numbers, e.g. $\pi, \mathrm{e}, 1, -0.5$
$[i..j]$	Set of integers $\{i, i+1, \dots, j-1, j\}$
$[i..j)$	Set of integers $\{i, i+1, \dots, j-1\}$

Table 2.2: Fundamental set definitions, provided that $i, j \in \mathbb{Z}$ and $i \leq j$.

$\mathtt{dom}(R) = \{a \in A \mid (a, b) \in R\}$ to denote the domain of R for at least one value a in A and correspondingly $\mathtt{rng}(R) = \{b \in B \mid (a, b) \in R\}$ to denote the range of R.

Elements of one set can be related to elements of the same or another set by means of a *relation* (see Definition 2.1.1). For example, we may say that an integer a relates to an integer b, provided that a is strictly less than b, i.e. $a < b$. In this case we have two objects $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$ that are related to each other by means of $<$, e.g. when $a = 3$ and $b = 8$. Table 2.3 briefly summarises common relations defined on sets that we shall use in this thesis.

Notation	Description
$A \times B$	Cross product, e.g. $\{1, 2\} \times \{a, b\} = \{(1, a), (1, b), (2, a), (2, b)\}$
$A \cap B$	Set intersection, e.g $\{1, 2\} \cap \{1, 3\} = \{1\}$
$A \cup B$	Set union, e.g. $\{1, 2\} \cup \{1, 3\} = \{1, 2, 3\}$
$A \setminus B$	Set difference, e.g. $\{1, 2\} \setminus \{1\} = \{2\}$
$A = B$	Set equality, e.g. $\{1, 2\} = \{2, 1\}$
$A \subseteq B$	Subset, e.g. $\{1, 2\} \subseteq \{2, 1\}$ and $\{1, 2\} \subseteq \mathbb{Z}$
$A \subset B$	Proper subset, e.g. $\{1\} \subset \{1, 2\}$

Table 2.3: Common set relations.

Array, matrix, and dictionary

For many of the described methodologies we use concrete object types that have specific properties. These include *array*, *matrix*, and *dictionary* objects. An array is a finite collection of elements of the same set, where each element is related to a unique index. Opposed to sets an array can contain the same element more than once, since they can be identified by their index. To denote an object a as an array we write $a = (a_0, a_1, \dots, a_{n-1}) = (a_i)_{i \in [0..n)}$, where $a_i \in A$ is an element of some set A. The length of the array object a is $|a| = n$. We use a_i to refer to the i-th element of a starting at index 0. We may also write $a[i] = a_i$ to express the same.

A matrix is a two-dimensional array whose elements are arranged in *columns* and *rows*. We call an object $\mathbf{A}$ with n columns and m rows an $n \times m$ matrix, which is expressed in box

bracket notation as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{1,0} & \cdots & \mathbf{A}_{n-1,0} \\ \mathbf{A}_{0,1} & \mathbf{A}_{1,1} & \cdots & \mathbf{A}_{n-1,1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{0,m-1} & \mathbf{A}_{1,m-1} & \cdots & \mathbf{A}_{n-1,m-1} \end{bmatrix}$$

The same can be expressed in a single term such that we may write $[\mathbf{A}_{i,j}]^{n \times m}$ to introduce the matrix $\mathbf{A}$, with $i \in [0..n)$ and $j \in [0..m)$. Unless specified otherwise, we assume that matrices defined in the remaining document use a column major-order representation. This means that $\mathbf{A}_{i,j}$, or equivalently $\mathbf{A}[i, j]$, represents the matrix element that intersects at the i-th column and the j-th row. Moreover, we may write $\mathbf{A}_{i,*} = (\mathbf{A}_{i,0}, \mathbf{A}_{i,1}, \ldots, \mathbf{A}_{i,m-1})$ to specify an array containing all elements that lie on the i-th column and correspondingly $\mathbf{A}_{*,j} = (\mathbf{A}_{0,j}, \mathbf{A}_{1,j}, \ldots, \mathbf{A}_{n-1,j})$ to specify all elements of the j-th row.

Lastly, we introduce a dictionary object type to denote an array whose elements are ordered by some ordering relation. More specifically, we assume that there is at least a strict weak order that allows to sort the elements in a dictionary $d = (d_i)_{i \in [0..n)}$ such that an element at index i compares either strictly less than to all other elements with an index higher than i (i.e. $d_i < d_j$) or is equivalent to them (i.e. neither $d_i < d_j$ nor $d_j < d_i$ is true) for all $j \in [i + 1..|d|)$. For example $d = (1, 2, 6, 16, 16, 16, 20, 99)$ represents a dictionary whose elements are sorted in ascending order and where the three elements $d_3 = d_4 = d_5 = 16$ correspond to the same equivalence class determined by the value 16. In this example these three elements satisfy the equality relation.

2.2 Sequences

The main subject of this thesis is to optimise algorithms that belong to the field of string processing, where the elements come from a special set called Σ.

Definition 2.2.1 (Alphabet)**.** An alphabet, denoted by the symbol Σ, is a non-empty, finite set over elements called *characters*, *letters*, or *symbols*.

The symbols of an alphabet represent categorical values that correspond to a specific domain. For example the DNA alphabet $\Sigma_{\text{DNA}} \coloneqq \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ represents the four fundamental DNA bases Adenine, Cytosine, Guanine, and Thymine, respectively.

Definition 2.2.2 (Sequence)**.** A *sequence* over Σ is a an array whose elements come from Σ. Thus, we formally describe a sequence x as $x = (x_0, x_1, \ldots, x_{n-1}) = (x_i)_{i \in [0..n)}$, with $x_i \in \Sigma$.

In the special case of sequences we may also describe a sequence as the *concatenation* of consecutive symbols and write $x = x_0x_1 \ldots x_{n-1} = x_0 + x_1 + \ldots + x_{n-1}$. In Table 2.4, we summarise common notations we will use to formally describe sequences and sets of sequences. Note that $\Sigma^0 = \{\epsilon\}$ and that $x_{j..i} = \epsilon$. Moreover, we may use the short notation $x \in \Sigma^n$ to properly introduce a sequence x of length $|x| = n$, with $x_i \in \Sigma$ for all $i \in [0..n)$.

We can transform a sequence into an integer array by means of a special *rank* function.

Notation	Description
$\|x\|$	Length of a sequence
ϵ	Empty sequence, i.e. $\|\epsilon\| = 0$
$x_{i..j}$	*Segment* or *infix* of x, i.e. $x_i x_{i+1} \dots x_{j-1}$; we may also write $x[i..j)$
$x_{0..j}$	*Prefix* of x; we may also write $x[0..j)$
$x_{i..n}$	*Suffix* of x; we may also write $x[i..n)$
Σ^n	Set of all finite sequences of length n over Σ
Σ^*	Set of all finite sequences over Σ, i.e. $\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$

Table 2.4: Common sequence notations, provided that $n \in \mathbb{N}_0$ and $0 \leq i \leq j \leq n$.

Definition 2.2.3 (Rank)**.** Let Σ be an alphabet, we define a special function, denoted by $\mathtt{rank}_\Sigma$, that maps each symbol of Σ to a distinct integer in the range $[0..|\Sigma|)$. Let a and b be two symbols from the same alphabet Σ, then it holds that $\mathtt{rank}_\Sigma(a) = \mathtt{rank}_\Sigma(b)$ if and only if $a = b$.

For the DNA alphabet Σ_{DNA} the ranks of the symbols could for example be $\mathtt{rank}_\Sigma(\mathtt{A}) = 0$, $\mathtt{rank}_\Sigma(\mathtt{C}) = 1$, $\mathtt{rank}_\Sigma(\mathtt{G}) = 2$, and $\mathtt{rank}_\Sigma(\mathtt{T}) = 3$. Thus, if we consider a DNA sequence `AGTACGACTAGCT` we can write this as $(0, 2, 3, 0, 1, 2, 0, 1, 3, 0, 2, 1, 3)$, which is the corresponding array of the associated ranks.

Using the rank representation, we can also compare two sequences x and y lexicographically.

Definition 2.2.4 (*lexicographic order*)**.** Given two sequences $x \in \Sigma^n$ and $y \in \Sigma^m$, with $n, m \in \mathbb{N}_0$. We say that $x <_{\mathrm{lex}} y$ is true if and only if, either $0 = n < m$, or $x_{0..k} = y_{0..k}$ and $\mathtt{rank}_\Sigma(x_k) < \mathtt{rank}_\Sigma(y_k)$, for some index $k \in [0.. \min n, m)$, provided that $[0.. \min n, m) \neq \emptyset$. In this case we write that x is lexicographically smaller than y.

In other words, k is the smallest index whose associated symbol rank in x is strictly less than the corresponding symbol rank in y, provided that the prefix of x equals the prefix of y up to that particular index. Furthermore, we have for two sequences $x, y \in \Sigma^* \setminus \{\epsilon\}$ that:

- $\epsilon \not<_{\mathrm{lex}} \epsilon$,
- $x \not<_{\mathrm{lex}} \epsilon$,
- $\epsilon <_{\mathrm{lex}} y$,
- $x <_{\mathrm{lex}} y$, if $x = y_{0..|x|}$, and
- $x \not<_{\mathrm{lex}} y$, if $y = x_{0..|y|}$.

That is x compares lexicographically less than y if x is a prefix of y.

2.3 Sequence alignment

The following section establishes fundamental concepts and algorithms related to the computation of pairwise sequence alignments. First we briefly discuss how alignments are represented and how they can be scored. This is followed by comparing various optimal pairwise alignment problems and how they can be computed using *dynamic programming* (DP).

2.3.1 Alignment representation

Sequence alignments are commonly used to identify differences between two or more sequences. In a biological sense, these differences indicate evolutionary events such as point mutations, insertions, or deletions. By identifying these events, one can, for example, draw conclusions about the relationships between the compared sequences, assign unknown sequences to particular families or groups etc. [Durbin et al., 1998]. If we consider only two sequences, we speak of a *pairwise sequence alignment*, whereas if we consider more than two sequences, we speak of a *multiple sequence alignment*. In the following, we will focus only on pairwise sequence alignment to describe the essential concepts.

Let $x \in \Sigma^n$ and $y \in \Sigma^m$ be two sequences over the same alphabet Σ. Conceptually, we represent a pairwise sequence alignment between x and y as a matrix with two rows. The first row represents the aligned sequence x, denoted by $\overline{x}$, and the second row the aligned sequence y, denoted by $\overline{y}$. We often use the terms *reference sequence* and *query sequence* to refer to the sequence assigned to the first and second row, respectively. Each column of this matrix represents an ordered pair of *aligned symbols*. These are symbols from the special *gapped alphabet* Σ_{gap}, which is defined as $\Sigma_{\text{gap}} = \Sigma \cup \{-\}$, where $-$ denotes a special space symbol that is not a member of Σ. This space symbol is used as a placeholder to adjust for the inserted and deleted symbols in the alignment, as is illustrated in Fig. 2.1.

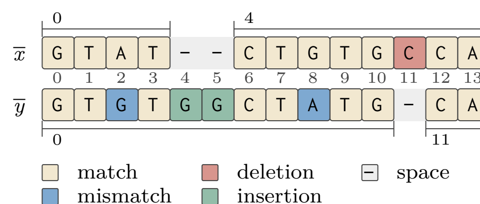


Figure 2.1: Visualisation of an alignment between a reference sequence $x =$ GTATCTGTGCCA and a query sequence $y =$ GTGTGGCTATGCA. Yellow tiles symbolise pairs of aligned positions with identical symbols (match). Blue tiles in the query sequence (position 2 and 8) represent replacements of the corresponding position in the reference sequence with another symbol (mismatch). Green tiles in the query sequence (position 4 and 5) mark inserted symbols, whereas red tiles in the reference sequence mark deleted symbols (position 11). Space symbols fill the resulting gaps in the opposite sequence. The crossbars represent the segments of the original sequences including their start position in the original sequence.

Definition 2.3.1 (Aligned sequence)**.** Let $s \in \Sigma^n$ be a some sequence. Furthermore, let $\pi_\Sigma : \Sigma_{\text{gap}}^* \rightarrow \Sigma^*$ denote a projection function that is recursively defined as:

$$\pi_\Sigma(\overline{s}) = \begin{cases} \epsilon & \text{if } \overline{s} = \epsilon, \\ \pi_\Sigma(\overline{s}') & \text{if } \overline{s} = \overline{s}' + -, \\ \pi_\Sigma(\overline{s}' + a) & \text{if } \overline{s} = \overline{s}' + a \text{ and } a \in \Sigma. \end{cases}$$

We say that $\overline{s}$ is an *aligned sequence* of a sequence s, if and only if $s = \pi_\Sigma(\overline{s})$. We call s the *source sequence* of $\overline{s}$.

Having this, we can now formally define a pairwise sequence alignment between x and y as follows.

Definition 2.3.2 (Pairwise sequence alignment)**.** A pairwise sequence alignment of the two source sequence $x \in \Sigma^n$ and $y \in \Sigma^m$ is an $N \times 2$ matrix $[\mathbf{A}_{i,j}]^{N \times 2}$, where:

- $\mathbf{A}_{i,j} \in \Sigma_{\text{gap}}$,
- $(\mathbf{A}_{i,0}, \mathbf{A}_{i,1}) \neq (-, -)$,
- $N \in [0..n + m)$,
- $\overline{x} = \mathbf{A}_{*,0}$ is an aligned sequence of x, and
- $\overline{y} = \mathbf{A}_{*,1}$ is an aligned sequence of y.

From this definition we can observe, that the relative order of the symbols of the source sequence is not changed when a sequence is aligned to another sequence.

Inside of an alignment $\mathbf{A}$ the aligned reference and query sequence have the same size. In the following we write $|\mathbf{A}| = N = |\overline{x}| = |\overline{y}|$. Moreover, we forbid the case that a column in an alignment $\mathbf{A}$ consists of two space symbols because it would not be biologically meaningful. Consequently, the maximal size of a pairwise sequence alignment is bounded by $n + m$. Considering a single column $\mathbf{A}_{i,*}$ of the alignment $\mathbf{A}$, we distinguish between:

- a *substitution* if and only if $\mathbf{A}_{i,0} \neq -$ and $\mathbf{A}_{i,1} \neq -$; we further use the term *match* if $\mathbf{A}_{i,0} = \mathbf{A}_{i,1}$ and *mismatch* otherwise.
- an *insertion* if and only if $\mathbf{A}_{i,0} = -$ and $\mathbf{A}_{i,1} \neq -$, and
- a *deletion* if and only if $\mathbf{A}_{i,0} \neq -$ and $\mathbf{A}_{i,1} = -$.

2.3.2 Alignment score

In many bioinformatic applications, the comparison of alignments plays a crucial role. However, directly comparing the representations of alignments would be impractical due to the vast number of possible pairwise alignments between two sequences of length n, which is given by $\binom{2n}{n}$ [Durbin et al., 1998]. Therefore, we employ a strategy to condense the alignment information into a single numeric value, as outlined in Definition 2.3.3, by utilising a *scoring scheme*.

Definition 2.3.3 (Scoring scheme)**.** Let $\sigma : \Sigma_{\text{gap}} \times \Sigma_{\text{gap}} \rightarrow \mathbb{R}$ be a score function that maps an aligned pair of symbols to some numeric value. Then the score of a pairwise sequence alignment $\mathbf{A}$ is computed using a scoring scheme, denoted by ψ, as:

$$\psi(\mathbf{A}) = \sum_{i=0}^{|\mathbf{A}|-1} \sigma(\mathbf{A}_{i,0}, \mathbf{A}_{i,1}).$$

A scoring scheme allows us to assign scores to different alignment elements, such as matches, mismatches, and spaces, providing a quantitative measure of the alignment quality and similarity. In different experimental settings, dedicated scoring schemes can be employed to ensure that the obtained alignment score reflects the desired interpretation of the alignments. Among these, the *edit scoring scheme* [Levenshtein, 1966] is one of the simpler ones, which calculates the absolute number of mismatches and spaces in the alignment.

Definition 2.3.4 (*Edit score function*)**.** Let $(a, b) \in \Sigma_{\text{gap}} \times \Sigma_{\text{gap}} \setminus \{-, -\}$ be two symbols from the same alphabet. The edit score function, denoted by σ_{edit}, is defined as:

$$\sigma_{\text{edit}}(a, b) = \begin{cases} 0 & \text{if } a = b, \\ -1 & \text{otherwise.} \end{cases}$$

For instance, considering the alignment depicted in Figure 2.1, the edit scoring scheme would yield a score of -5, taking into account the five mismatches and spaces present in the alignment.

While the edit scoring scheme is commonly used in applications like read mapping, where error quantification is important, it may not be suitable for studying the ancestral relationships between aligned sequences. In such cases, accurately characterising true mutational events that occurred during sequence evolution becomes crucial. For instance, an InDel (Insertion, Deletion) is considered a single evolutionary event, as it is unlikely that a consecutive stretch of inserted or deleted symbols results from multiple independent events occurring repeatedly at the same site in the DNA sequence by chance. To address this, certain applications utilise a generalised scoring scheme that distinguishes between substitution and InDel events, allowing for a more accurate analysis of ancestral relationships.

Substitution score function

The *substitution score function* focuses solely on evaluating substitutions within an alignment. There are two commonly employed substitution score functions: the *unitary score function* and the *matrix score function*. We will begin by providing the definition of the unitary score function.

Definition 2.3.5 (Unitary score function)**.** Let $a \in \Sigma$ and $b \in \Sigma$ be two symbols from the same alphabet, excluding the space symbol $-$ (i.e. $- \notin \Sigma$). We introduce two parameters: $m_= \in \mathbb{N}_0$, representing the scores for a match, and $m_{\neq} \in \{\dots, -2, -1\}$, representing the scores for a mismatch. The unitary score function, denoted by σ_{uni}, is defined as:

$$\sigma_{\text{uni}}(a, b) = \begin{cases} m_= & \text{if } a = b, \\ m_{\neq} & \text{otherwise (i.e.} a \neq b). \end{cases}$$

The unitary score function is a more general form of the edit scores, allowing for arbitrary positive score values for matches and negative score values strictly less than zero for mismatches. This function is commonly used in aligning DNA sequences. In contrast, the matrix score function enables the assignment of a distinct score value to each combination of aligned symbols.

Definition 2.3.6 (Matrix score function)**.** Let $a \in \Sigma$ and $b \in \Sigma$ be two symbols from the same alphabet Σ, and $l = |\Sigma|$ be the size of the alphabet. Additionally, let $[\mathbf{M}_{i,j}]^{l \times l}$ be a matrix, where $\mathbf{M}_{i,j} \in \mathbb{Z}$ represents the score value for aligning symbols a and b. Then, we define the matrix score function, denoted by σ_{mtx}, as:

$$\sigma_{\text{mtx}}(a, b) = \mathbf{M}_{i,j},$$

with $i = \texttt{rank}_\Sigma(a)$ and $j = \texttt{rank}_\Sigma(b)$.

Two well-known examples of matrix score functions used for assessing alignments between amino acid sequences are the BLOSUM (blocks substitution matrix) [Henikoff and Henikoff, 1992] and PAM (point accepted mutation matrix) [Dayhoff et al., 1978] matrices. These matrices are specifically designed to capture the physical and chemical characteristics of amino acids. They take into account the likelihood of substituting one amino acid with another that shares similar traits, as opposed to replacing it with an amino acid that could potentially disrupt protein function.

The score values within these matrices reflect the probabilities of specific amino acid substitutions observed in pairwise alignments of sequences with a known degree of sequence similarity. These probabilities are typically calculated using log-odds ratios, indicating the likelihood of observing a particular aligned pair of amino acids compared to its expected frequency. Higher values in the matrix indicate a higher probability of aligning the corresponding pair of amino acids, while lower values indicate the opposite. For example, the BLOSUM62 score matrix is derived from pairwise alignments of protein sequences that exhibit a minimum sequence similarity of 62% (refer to Fig. 2.2).

	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
A	4	0	-2	-1	-2	0	-2	-1	-1	-1	-1	-2	-1	-1	-1	1	0	0	-3	-2
C	0	9	-3	-4	-2	-3	-3	-1	-3	-1	-1	-3	-3	-3	-3	-1	-1	-1	-2	-2
D	-2	-3	6	2	-3	-1	-1	-3	-1	-4	-3	1	-1	0	-2	0	-1	-3	-4	-3
E	-1	-4	2	5	-3	-2	0	-3	1	-3	-2	0	-1	2	0	0	-1	-2	-3	-2
F	-2	-2	-3	-3	6	-3	-1	0	-3	0	0	-3	-4	-3	-3	-2	-2	-1	1	3
G	0	-3	-1	-2	-3	6	-2	-4	-2	-4	-3	0	-2	-2	-2	0	-2	-3	-2	-3
H	-2	-3	-1	0	-1	-2	8	-3	-1	-3	-2	1	-2	0	0	-1	-2	-3	-2	2
I	-1	-1	-3	-3	0	-4	-3	4	-3	2	1	-3	-3	-3	-3	-2	-1	3	-3	-1
K	-1	-3	-1	1	-3	-2	-1	-3	5	-2	-1	0	-1	1	2	0	-1	-2	-3	-2
L	-1	-1	-4	-3	0	-4	-3	2	-2	4	2	-3	-3	-2	-2	-2	-1	1	-2	-1
M	-1	-1	-3	-2	0	-3	-2	1	-1	2	5	-2	-2	0	-1	-1	-1	1	-1	-1
N	-2	-3	1	0	-3	0	1	-3	0	-3	-2	6	-2	0	0	1	0	-3	-4	-2
P	-1	-3	-1	-1	-4	-2	-2	-3	-1	-3	-2	-2	7	-1	-2	-1	-1	-2	-4	-3
Q	-1	-3	0	2	-3	-2	0	-3	1	-2	0	0	-1	5	1	0	-1	-2	-2	-1
R	-1	-3	-2	0	-3	-2	0	-3	2	-2	-1	0	-2	1	5	-1	-1	-3	-3	-2
S	1	-1	0	0	-2	0	-1	-2	0	-2	-1	1	-1	0	-1	4	1	-2	-3	-2
T	0	-1	-1	-1	-2	-2	-2	-1	-1	-1	-1	0	-1	-1	-1	1	5	0	-2	-2
V	0	-1	-3	-2	-1	-3	-3	3	-2	1	1	-3	-2	-2	-3	-2	0	4	-3	-1
W	-3	-2	-4	-3	1	-2	-2	-3	-3	-2	-1	-4	-4	-2	-3	-3	-2	-3	11	2
Y	-2	-2	-3	-2	3	-3	2	-1	-2	-1	-1	-2	-3	-1	-2	-2	-2	-1	2	7

11 highest, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4 lowest

Figure 2.2: Cost values of the BLOSUM62 score matrix for the 20 amino acids represented by their one letter IUPAC (International Union of Pure and Applied Chemistry) code.

Gap score function

To accurately assess InDels, a specific score function called the *gap score function* is utilised.

Definition 2.3.7 (Gap)**.** A *gap* is the longest consecutive stretch of spaces within an aligned sequence.

The gap score function plays a crucial role in accurately evaluating InDels within sequence alignments. It assigns a specific score value to each gap based on its length, allowing for a more comprehensive assessment of InDel events. In general, there are different approaches to modelling gaps, with two commonly used ones being the *affine gap function* and the *linear gap function*.

Definition 2.3.8 (Affine gaps)**.** Let $l \in \mathbb{N}_0$ denote the length of a gap. We introduce two parameters: $g_{\text{opn}} \in \{\ldots, -2, -1, 0\}$, representing the negative scores for opening a gap, and $g_{\text{ext}} \in \{\ldots, -2, -1, 0\}$, representing the negative scores for extending an already opened gap. The affine gap function, denoted by γ_{aff}, is defined as

$$\gamma_{\text{aff}}(l) = \begin{cases} g_{\text{opn}} + l \cdot g_{\text{ext}} & \text{if and only if } l > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 2.3.9 (Linear gaps)**.** We derive the linear gap function, denoted by γ_{lin}, from the affine gap score function (see Definition 2.3.8) if we set $g_{\text{opn}} = 0$, such that $\gamma_{\text{lin}}(l) = l \cdot g_{\text{ext}}$.

In both cases, the total score of a gap decreases linearly with the length of the gap, while in the case of affine gaps an initial penalty is added combining nearby gaps into a single InDel event [Cartwright, 2006]. By incorporating these different gap score functions into the scoring scheme, we can appropriately capture the characteristics of InDel events, enabling a more nuanced analysis of sequence alignments (refer to Fig. 2.3).

(a) Linear gaps

(b) Affine gaps

Figure 2.3: Effects of choosing either linear (a) or affine gap score functions (b). The three single deletion events in a are condensed into a single deletion event of size three in b.

2.3.3 Optimal pairwise alignment

A fundamental problem occurring in numerous bioinformatic applications is the computation of the *optimal alignment* for a given pair of sequences, which we define as follows.

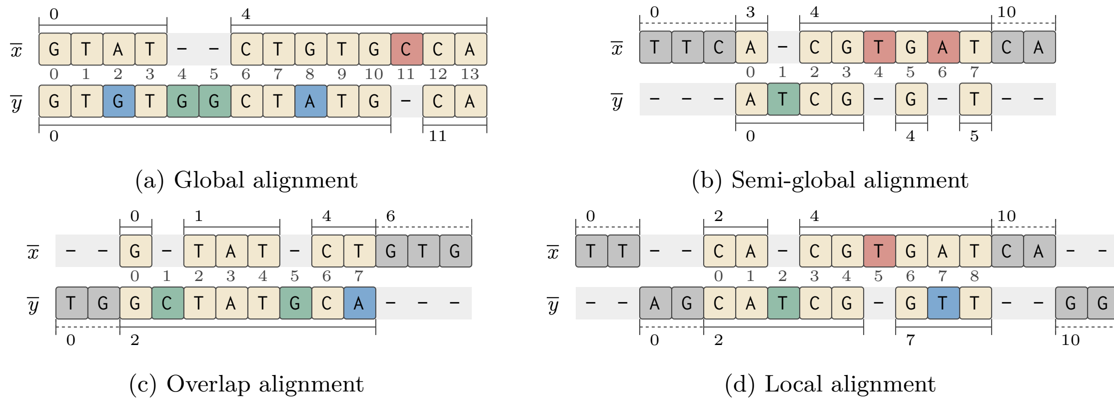
Definition 2.3.10 (Optimal pairwise alignment problem)**.** Let $\mathcal{A}(x, y) \subset \Sigma^*_{\text{gap}} \times \Sigma^*_{\text{gap}}$ denote the set of all proper alignments between x and y. Then, given a scoring scheme ψ, the optimal pairwise alignment problem is to find the set

$$\Lambda(x, y : \psi) = \underset{\mathbf{A} \in \mathcal{A}(x,y)}{\arg\max}\, \psi(\mathbf{A}), \tag{2.1}$$

for which the alignment score is maximal.

The general problem of finding the optimal alignment can be further divided into subclasses. Among these the *optimal global alignment*, the *optimal semi-global alignment*, the *optimal overlap alignment*, and the *optimal local alignment* are well-known representatives. The main difference between these subclasses is that they limit the set of valid alignments between the two sequences x and y.

The global alignment considers all alignments that align the reference sequence to the query sequence from end-to-end. According to Definition 2.3.1, we require that $x = \pi_\Sigma(\mathbf{A}_{*,0})$ and $y = \pi_\Sigma(\mathbf{A}_{*,1})$ for all $\mathbf{A} \in \mathcal{A}(x, y)$ (see Fig. 2.4a). The semi-global alignment relaxes this condition to allow any alignments between any segment of the reference sequence that is aligned to the whole query sequence, i.e. $y = \pi_\Sigma(\mathbf{A}_{*,1})$ (see Fig. 2.4b). In the case of the overlap alignment, only alignments between any prefix of the reference sequence with any suffix of the query sequence or vice versa are allowed (see Fig. 2.4c). Lastly, the local alignment considers all alignments between any segment of the reference sequence with any segment of the query sequence (see Fig. 2.4d).



(a) Global alignment (b) Semi-global alignment

(c) Overlap alignment (d) Local alignment

Figure 2.4: Examples of global (a), semi-global (b), overlap (c), and local alignment (d) for distinct sequence pairs. Symbols with grey background are unaligned and not part of the corresponding alignment.

2.3.4 Dynamic programming

The optimal alignment problem can be solved systematically using a DP approach. The general idea is to recursively break down the original problem of finding the optimal alignment between two sequences into manageable subproblems. The final solution is then built from the partial solutions obtained during the recursion.

To be more precise, let us consider the problem of finding the optimal global alignment between $x \in \Sigma^n$ and $y \in \Sigma^m$. At each step of the recursion, we compute the score of the optimal alignment between the two prefixes $x_{0..i}$ and $y_{0..j}$, where $i \in [1..n]$ and $j \in [1..m]$. There are three possibilities how the last symbols x_{i-1} and y_{j-1} of the two prefixes could have been aligned: by substituting x_{i-1} with y_{j-1}, by deleting x_{i-1}, or by inserting y_{j-1}. To determine which one of these three possibilities is the optimal choice, we consider the alignment score of the optimal alignment that ended immediately before the respective events and add the scores for the corresponding operation. Figure 2.5 gives a visual impression of this recursive approach.

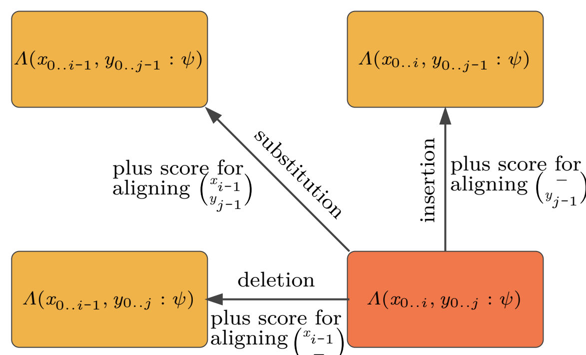


Figure 2.5: General recursion dependency of the dynamic programming algorithm.

Alignment algorithm

To efficiently compute the alignment score, the alignment algorithm utilises a dedicated *DP matrix* $[\mathbf{S}_{i,j}]^{n+1 \times m+1}$. The DP matrix has $n+1$ columns and $m+1$ rows. The additional column and row serve as a recursion anchor, and their initialisation depends on the alignment subclass being computed.

The *DP algorithm* can be divided into three phases: *initialisation*, *recursion*, and *finalisation*. In the initialisation phase, the first column and row of the DP matrix $\mathbf{S}$ are filled according to the specific requirements of the alignment subclass being computed. During the recursion phase, the entire DP matrix is computed in a major column order, following the procedure illustrated in Fig. 2.6. At each cell (i, j), the maximum value among the three possible options for ending the alignment at that position is computed using the respective recursion formula (see Eq. (2.4) using a linear gap function and Eq. (2.10) using a affine gap function). In the finalisation phase, the *DP coordinate* $(\hat{\imath}, \hat{\jmath})$, with $\hat{\imath} \in [0..n]$ and $\hat{\jmath} \in [0..m]$, that corresponds to the maximum score value in $\mathbf{S}$ is determined and returned as the result of the alignment algorithm.

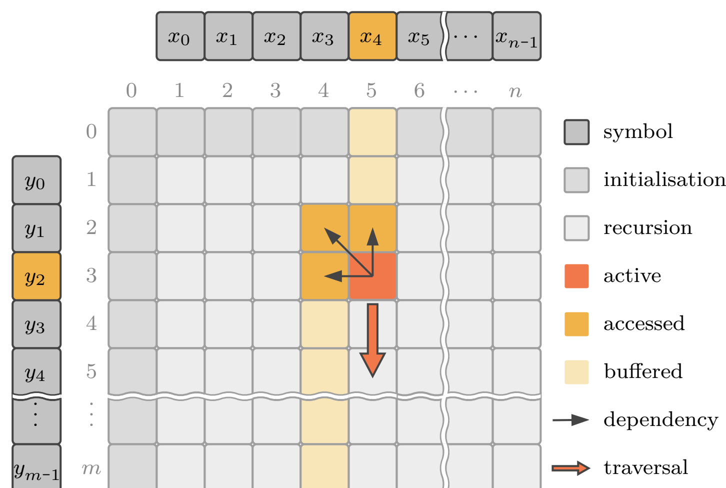


Figure 2.6: Computing the DP matrix for sequences x and y. Currently, entry at position $(5, 3)$ is computed (active entry) by accessing the entries at $(4, 2)$ (diagonal arrow), $(4, 3)$ (horizontal arrow), and $(5, 2)$ (vertical arrow) as well as the sequence symbols x_4 and y_2. The optimal alignment score can be computed with a linear buffer using in total $m+1$ cells to cache the scores of the previous column(orange and yellow tiles).

The implementation details of the DP algorithm may vary depending on the specific alignment problem being solved and the chosen gap score function. Here, we will focus on the differences in computing the different alignment problems assuming the use of linear gaps.

Global alignment To compute the optimal global alignment [Needleman and Wunsch, 1970], the first row and column of $\mathbf{S}$ are initialised with the scores for a gap of length i and

j respectively.

$$\mathbf{S}_{i,0} = i \cdot g_{\text{ext}} \tag{2.2}$$
$$\mathbf{S}_{0,j} = j \cdot g_{\text{ext}} \tag{2.3}$$

During the recursion the alignment score for the currently visited cell $\mathbf{S}_{i,j}$ is computed by

$$\mathbf{S}_{i,j} = \max \begin{cases} \mathbf{S}_{i-1,j-1} + \sigma(x_{i-1}, y_{j-1}) & \text{(Substitution)} \\ \mathbf{S}_{i-1,j} + g_{\text{ext}} & \text{(Deletion)} \\ \mathbf{S}_{i,j-1} + g_{\text{ext}} & \text{(Insertion),} \end{cases} \tag{2.4}$$

with $i \in [1..n]$ and $j \in [1..m]$. The optimal alignment score is received from the last DP coordinate of the matrix, i.e. $\hat{i} = n$ and $\hat{j} = m$.

Semi-global alignment Computing the optimal semi-global alignment only differs from the global alignment by setting all scores of the first row to 0, i.e. $\mathbf{S}_{i,0} = 0$, and by extending the search for the optimal alignment score to the entire last row in the finalisation phase, i.e. $\hat{i} = \max(\mathbf{S}_{*,m})$ and $\hat{j} = m$. Due to these adaptions, leading and trailing gaps in $\overline{x}$ are not penalised, such that we can find the optimal alignment between the whole query sequence y and any segment of the reference sequence x.

Overlap alignment Computing the optimal overlap alignment further refines the initialisation of the DP matrix and the search space. In this mode, both the cells of the first column and row are initialised with 0, i.e. $\mathbf{S}_{0,j} = \mathbf{S}_{i,0} = 0$. Furthermore, the maximum score value is searched in the last row and column, i.e. $\hat{i} = \max(\mathbf{S}_{*,m})$ and $j = m$ or $\hat{j} = \max(\mathbf{S}_{n,*})$ and $\hat{i} = n$. Correspondingly, the optimal alignment can start anywhere in the first row or column and can end anywhere in the last row or column which models the desired behaviour of aligning any prefix of x with any suffix of y or vice versa.

Local alignment To compute an optimal local alignment, the same initialisation strategy as described for the overlap alignment is used. In addition, the main recursion formula is refined to forbid any alignment score below 0 [Smith and Waterman, 1981], such that

$$\mathbf{S}_{i,j} = \max \begin{cases} \mathbf{S}_{i-1,j-1} + \sigma(x_{i-1}, y_{j-1}) & \text{(Substitution)} \\ \mathbf{S}_{i-1,j} + g_{\text{ext}} & \text{(Deletion)} \\ \mathbf{S}_{i,j-1} + g_{\text{ext}} & \text{(Insertion)} \\ 0, & \end{cases} \tag{2.5}$$

In the finalisation phase the entire space of the DP matrix $\mathbf{S}$ is searched for the maximal alignment score, i.e. $(\hat{i}, \hat{j}) = \arg\max_{i,j} \mathbf{D}_{i,j}$. This is typically achieved by tracking the maximum score while computing the DP matrix. By choosing 0 as the lower bound for the score values and considering the entire space of the DP matrix, we can compute the optimal alignment between any segment of x and any segment of y.

Using affine gaps

To compute optimal alignments using an affine score function, Gotoh [1990] introduced two auxiliary DP matrices with the same dimensions as the main DP matrix $\mathbf{S}$ to cache the

score of the optimal alignment that ends in a deletion or insertion. Let $[\mathbf{D}_{i,j}]^{n+1\times m+1}$ and $[\mathbf{I}_{i,j}]^{n+1\times m+1}$ be the corresponding score matrices for deletions and insertions respectively. The initialisation of the global alignment (see Eq. (2.3)) changes to

$$\begin{aligned}\mathbf{S}_{i,0} &= g_{\text{opn}} + i \cdot g_{\text{ext}}\\ \mathbf{D}_{i,0} &= g_{\text{opn}} + i \cdot g_{\text{ext}}\\ \mathbf{I}_{i,0} &= -\infty\end{aligned} \tag{2.6}$$

for the initial row and

$$\begin{aligned}\mathbf{S}_{0,j} &= g_{\text{opn}} + j \cdot g_{\text{ext}}\\ \mathbf{D}_{0,j} &= -\infty\\ \mathbf{I}_{0,j} &= g_{\text{opn}} + j \cdot g_{\text{ext}}\end{aligned} \tag{2.7}$$

for the initial column. Note by initialising the first row of $\mathbf{I}$ and the first column of $\mathbf{D}$ to $-\infty$ we enforce that a leading gap in either sequence is penalised with the scores for open a gap.

The main recursion formula for the global alignment from Eq. (2.4) is then updated to

$$\mathbf{D}_{i,j} = \max\begin{cases}\mathbf{D}_{i-1,j} + g_{\text{ext}}\\ \mathbf{S}_{i-1,j} + g_{\text{opn}} + g_{\text{ext}}\end{cases} \tag{2.8}$$

$$\mathbf{I}_{i,j} = \max\begin{cases}\mathbf{I}_{i,j-1} + g_{\text{ext}}\\ \mathbf{S}_{i,j-1} + g_{\text{opn}} + g_{\text{ext}}\end{cases} \tag{2.9}$$

$$\mathbf{S}_{i,j} = \max\begin{cases}\mathbf{S}_{i-1,j-1} + \sigma(x_{i-1}, y_{j-1}) & \text{(substitution)}\\ \mathbf{D}_{i,j} & \text{(deletion)}\\ \mathbf{I}_{i,j} & \text{(insertion).}\end{cases} \tag{2.10}$$

Trace algorithm

To obtain the actual alignment representation, we annotate the DP matrix with special symbols that indicate the direction from which the maximum at a coordinate (i, j) was derived. These trace markers are defined by the *trace alphabet*, denoted as $\Sigma_{\text{trace}} = \{\nwarrow, \leftarrow, \uparrow, \Leftarrow, \Uparrow, \circlearrowleft\}$. When read from left to right, these symbols represent the possible trace directions: substitution ($\nwarrow$), deletion ($\leftarrow$), opening a deletion ($\Leftarrow$), insertion ($\uparrow$), opening an insertion ($\Uparrow$), and a terminal marker ($\circlearrowleft$) indicating the end of the trace operation. To facilitate this, we allocate an additional *trace matrix* $\mathbf{T} = [\mathbf{T}_{i,j}]^{n+1\times m+1}$ over the trace alphabet. The first coordinate, $\mathbf{T}_{0,0}$, is initialised as $\circlearrowleft$. The initialisation of the first row and column depends on the specific initialisation rule and the gap score function being used. If a value in the first row or column is explicitly set to 0, the corresponding coordinate in Tis initialised as $\circlearrowleft$. Otherwise, we set the entries of the first row to $\leftarrow$ and the entries of the first column to $\uparrow$. If affine gaps are used, then $\mathbf{T}_{1,0}$ is set to $\Leftarrow$, and $\mathbf{T}_{0,1}$ is set to $\Uparrow$. During the recursion phase, we select the trace marker that corresponds to the path for which the maximum score value was computed. In the case of the local alignment, if the maximum score is 0, the trace marker is set to $\circlearrowleft$.

To construct the actual alignment, we start with empty aligned sequences, i.e. $\overline{x} = \epsilon$ and $\overline{y} = \epsilon$. Then, we begin at the coordinate $(\hat{\imath}, \hat{\jmath})$ that corresponds to the found optimal alignment score and go to the next coordinate indicated by the read trace symbol. Let (i, j) be the currently visited coordinate, we can describe this as a function `trace` defined as:

$$\texttt{trace}(\mathbf{T}_{i,j}) = \begin{cases} \mathbf{T}_{i-1,j-1} & \text{if } \mathbf{T}_{i,j} = \nwarrow, \\ \mathbf{T}_{i,j-1} & \text{if } \mathbf{T}_{i,j} \in \{\uparrow, \Uparrow\}, \\ \mathbf{T}_{i-1,j} & \text{if } \mathbf{T}_{i,j} \in \{\leftarrow, \Leftarrow\}, \text{ or} \\ \mathbf{T}_{i,j} & \text{if } \mathbf{T}_{i,j} = \circlearrowleft. \end{cases}$$

For each visited trace symbol along the trace path, we incrementally build the aligned sequences by

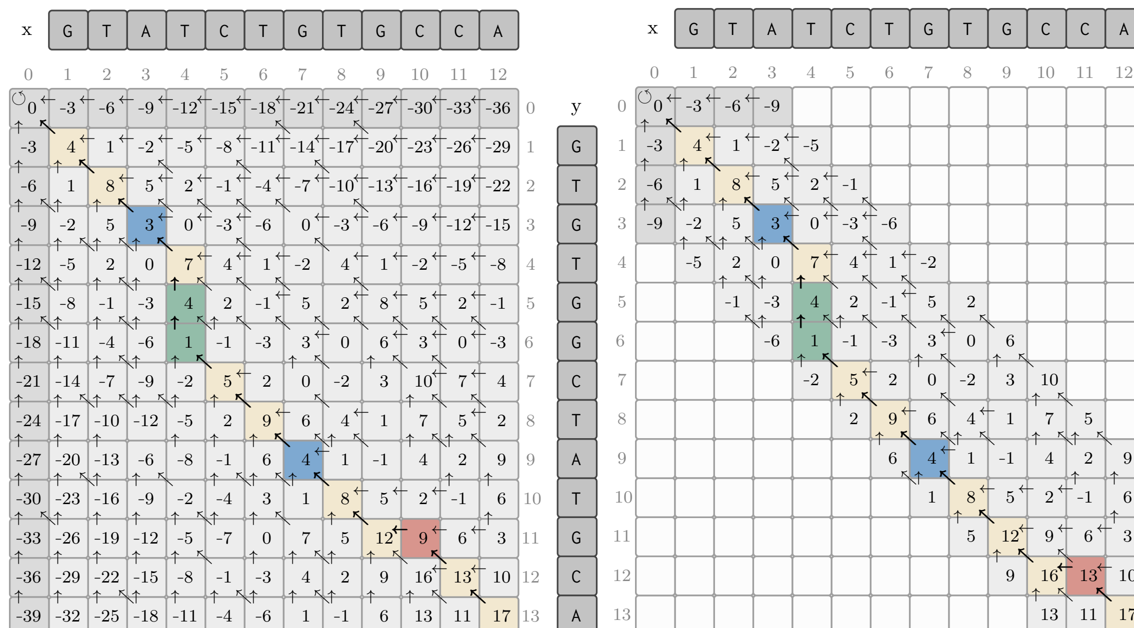
- $\overline{x} \leftarrow x_{i-1}\overline{x}$ and $\overline{y} \leftarrow y_{j-1}\overline{y}$ if $\mathbf{T}_{i,j} = \nwarrow$,
- $\overline{x} \leftarrow -\overline{x}$ and $\overline{y} \leftarrow y_{j-1}\overline{y}$ if $\mathbf{T}_{i,j} \in \{\uparrow, \Uparrow\}$, or
- $\overline{x} \leftarrow x_{i-1}\overline{x}$ and $\overline{y} \leftarrow -\overline{y}$ if $\mathbf{T}_{i,j} \in \{\leftarrow, \Leftarrow\}$.

In the case where $\mathbf{t}_{i,j} = \circlearrowleft$, the trace procedure is stopped.

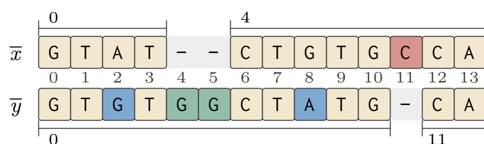
Sometimes, multiple alternative traces can lead to the same optimal alignment score. In such cases, the trace procedure prioritises a specific direction when following the trace symbols (typically in the order of 1. $\nwarrow$, 2. $\leftarrow$ or $\Leftarrow$, and 3. $\uparrow$ or $\Uparrow$). The first alignment obtained by following the trace is referred to as the optimal alignment (see Fig. 2.7b), while the alternative alignments are called *cooptimal alignments*. For instance, in the alignment example shown in Fig. 2.7a, the trace corresponds to the optimal alignment depicted in Fig. 2.7b. However, in the case of the banded alignment, the trace leads to a cooptimal alignment as illustrated in Fig. 2.7c. Both alignments represent a global alignment with a maximum alignment score of 17, but the position of the deletion can vary, occurring either at position 11 or 12 of the final alignment.

Banded algorithm

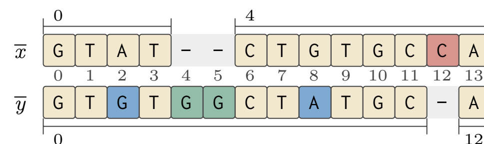
It is also possible to compute only a subset of the original DP matrix by defining a concentric band with a predefined width k around the main diagonal. In this case we speak of computing a *banded alignment*. The main diagonal extends from the top left corner to the bottom right corner of the DP matrix. If the sequences have different lengths, the main diagonal ends either to the left of or above the bottom right corner in the last row or last column of the DP matrix, respectively. The k-band includes all entries of the DP matrix that are $\left\lceil \frac{k}{2} \right\rceil$ diagonals below and $\left\lfloor \frac{k}{2} \right\rfloor$ diagonals to the right of the main diagonal, including the main diagonal itself. In other words, there are $k+1$ main diagonals within the band. During the recursion, only the entries within this band are computed, while all other entries of the DP matrix are skipped. The right image in Figure 2.7a illustrates this concept using a band of size $k = 6$.



(a) Filled DP matrix using the full space (left) and a band of size 6 (right).



(b) The optimal alignment.



(c) A cooptimal alignment.

Figure 2.7: Example of a computed DP matrix (a, left) and its banded version (a right) to compute the optimal global alignment between $x =$ GTATCTGTGCCA and $y =$ GTGTGGCTATGCA using unitary substitution cost function, with $m_{=} = 4$ and $m_{\neq} = -5$, and linear gap score function, with $g_{\text{ext}} = -3$. The score of the optimal alignment is 17 and is found at coordinate $(12, 13)$. Arrows represent the trace marker. Bold arrows mark the path of the optimal (b), respectively cooptimal alignment (c).

Complexity analysis

The regular DP algorithm visits each entry of the score matrix exactly once, regardless of the alignment problem being solved. Assuming that the scoring scheme function σ is constant, the algorithm performs a constant number of instructions to update the current alignment score in each iteration, regardless of the gap model being used. As a result, the total runtime of the regular DP algorithm is $\mathcal{O}(nm)$.

By keeping only a single column of the score matrix in memory, we can compute the alignment score in linear space $\mathcal{O}(m)$ (refer to Fig. 2.6). However, to compute the actual alignment, we need to fill the entire trace matrix in each iteration of the recursion, which requires a constant number of comparison and assignment instructions. Therefore, the runtime complexity remains at $\mathcal{O}(nm)$ in the worst case. Additionally, compared to computing only the scores, we need to allocate memory proportional to the product of both sequence lengths, i.e. $\mathcal{O}(nm)$. Following the trace path to obtain the alignment transcript, on the

other hand, requires at most $\mathcal{O}(n+m)$ steps.

In some cases, when we know that the aligned sequences are similar, we can execute the DP algorithm with a k-band, which reduces the runtime complexity to $\mathcal{O}(kn)$. Furthermore, the total space consumption of both the score and trace matrix is reduced to $\mathcal{O}(kn)$.

2.4 Pattern matching

The second major class of algorithms we consider in this work are pattern matching algorithms [Gusfield, 1997]. The pattern matching problem aims to identify all distinct locations, referred to as *k-occurrences*, at which a query sequence $y \in \Sigma^m$ occurs within a reference sequence $x \in \Sigma^n$, allowing for at most k errors [Galil and Giancarlo, 1988]. They are especially useful when searching for recurring patterns or motifs within a larger sequence such that we assume that $m \ll n$.

Definition 2.4.1 (k-occurrence)**.** Given $k \in \mathbb{N}_0$ and an alignment $\mathbf{A} \in \mathcal{A}(x', y)$ between a segment x' of x and y. We call x' a k-occurrence if and only if $\psi(\mathbf{A} : \sigma_{\text{edit}}) \geq -k$.

In the case where $k > 0$ we speak of *approximate pattern matching* and when $k = 0$ we speak of *exact pattern matching*.

The algorithms used to solve these problems often employ techniques such as dynamic programming, finite automata, or data structures like suffix trees or indexing methods to efficiently identify the k-occurrences of the query sequence within the reference sequence. In the following, we will first examine representative examples of *online pattern matching* algorithms and then proceed to discuss *filtration* approaches that utilise an additional *index* structure. For a detailed survey of this subject, the interested reader is referred to [Navarro, 2001].

2.4.1 Online methods

To solve the pattern matching problem algorithmically, a common approach is to shift a *window* of size $\omega = m + k$ from left to right over the reference sequence. At each position, an algorithm-specific *predicate* is invoked. This function reduces the given input to a boolean value, either **true** or **false**, by comparing the query sequence with the segment x' of x that corresponds to the currently visited window, i.e. $|x'| = \omega$. When the predicate function returns **true**, the current start position of the window in the reference sequence is reported as a k-occurrence.

Exact pattern matching

The naïve solution for computing the set of all exact occurrences is presented in Algorithm 2.1. The main loop, which shifts the window of size $\omega = m$ over the reference sequence, is depicted in lines 9 to 11. During each iteration, the predicate `isEqual` (lines 1 to 7) is called with the corresponding segment $x' = x_{i..i+\omega}$ of the currently visited window starting at index i and the query sequence. The predicate compares the two sequences symbol by symbol, and

only if all symbols are equal, it returns **true**. The runtime complexity of this algorithm is $\mathcal{O}(nm)$ in the worst case.

Algorithm 2.1: Naïve search

```
Input: x, y
1  isEqual(x', y):
2      if |x'| ≠ |y| then
3          return false
4      for i ← 0 to |x| − |y| do
5          if x'_i ≠ y_i then
6              return false
7      return true
8  ω ← |y|
9  for i ← 0 to n − ω do
10     if isEqual(x_{i..i+ω}, y) then
11         Report occurrence at i
```

Various strategies have been proposed to optimise the runtime of the naïve solution, either by being faster on average or by improving the worst-case runtime [Gusfield, 1997; Navarro, 1998]. These strategies achieve their improvements by adding a preprocessing step, in which they construct auxiliary data structures for the query sequence before performing the search. We will briefly introduce two representative algorithms: the *Boyer-Moore-Horspool*, or simply *Horspool* algorithm [Horspool, 1980], and the *Bitap* algorithm, also known as the *ShiftOR* algorithm [Baeza-Yates, 1989].

Horspool Instead of comparing the query sequence at every position of the reference sequence, Horspool [1980] devised a practical solution that skips text positions where it is guaranteed that the predicate cannot evaluate to **true**. Their approach involves preprocessing y in $\mathcal{O}(m)$ time using $\mathcal{O}(|\Sigma|)$ additional space to create a lookup array. This array stores, for each symbol of the alphabet, the maximum number of steps the window can be skipped in the next iteration.

Similar to the naïve approach, the main search algorithm scans linearly over the reference sequence from left to right and compares the query sequence with the segment of the reference sequence corresponding to the current search window. However, after each invocation of the predicate, the algorithm jumps to the next reference sequence position using the precomputed lookup array. While its worst-case runtime is still $\mathcal{O}(nm)$, it has been shown that this strategy outperforms the naïve search on average for query sizes larger than 5 [Horspool, 1980].

ShiftOr [Baeza-Yates, 1989] improved the worst-case runtime to $\mathcal{O}(\lceil \frac{m}{w} \rceil\, n)$, where w represents the number of bits of the underlying computer word (typically 64). This improvement was achieved by constructing a lookup array of size $|\Sigma|$ that contains query-specific bitmasks for each symbol present in the query sequence. The construction of this array takes $\mathcal{O}(m + |\Sigma|)$ time.

During the main search, the algorithm scans linearly over the reference sequence and updates the bits in an additional control mask using the corresponding bitmask from the preprocessing for the active symbol in the currently visited search window over x. After each update, the algorithm checks if the bit at position $m-1$ is set in the control mask. If the check is successful, the algorithm reports an occurrence starting at position $i-m+1$.

During the execution of the for-loop, the algorithm processes w symbols of the query sequence simultaneously using bit-parallel instructions. This allows the algorithm to search y in x in linear time, assuming $m \leq w$. The algorithm can be generalised to handle query sequences of arbitrary length, resulting in a runtime of $\lceil \frac{m}{w} \rceil$.

Approximate pattern matching

To solve the approximate pattern matching problem, we could theoretically use the standard DP algorithm for computing semi-global alignments, as described in the previous section. However, in practice, we are only interested in counting errors, so we can use a more efficient DP algorithm implementation proposed by Myers [1999]. This algorithm is an adaptation of the general DP algorithm (see Section 2.3.4) used to solve the global alignment problem. It utilises bit-parallel instructions during the recursion of the DP matrix, taking advantage of the special properties of the edit scoring scheme (see Definition 2.3.4).

Myers showed that there are only three possible ways in which the scores of two adjacent cells in vertical, horizontal, or diagonal directions can differ. Based on this observation, he encoded the relative differences of the alignment scores in the scoring matrix using a set of six bitvectors. These bitvectors are used to update an entire column of the DP matrix using only constant number of bit-parallel instructions. This yielded an algorithm computing the edit distance in $\mathcal{O}(\frac{nm}{w} + m \cdot |\Sigma|)$ using only $\mathcal{O}(\frac{m}{w} \cdot |\Sigma|)$ space.

Ukkonnen trick Ukkonen [1985] showed how the runtime could be further reduced when the maximum number of allowed errors k is known in advance. He observed that once the algorithm reaches a score less than $-k$ in an active cell, all subsequent scores in the same column can not produce an optimal alignment since the score can not increase. Therefore, the algorithm can skip the computation of the remaining entries in the current column. This optimisation significantly reduces the overall runtime complexity to $\mathcal{O}(\frac{kn}{w} + m \cdot |\Sigma|)$.

2.4.2 Filter methods

Although there are various efficient online pattern matching algorithms available, they can become inefficient when dealing with large problem sizes, such as matching one million reads to a reference sequence during read mapping. To address this issue, current solutions employ a *filter* prior to the pattern matching process. This filter is designed to exclude positions in x where a k-occurrence between y and x is impossible.

The effectiveness of a filter is measured by its *specificity*, defined as:

$$\frac{\text{true negatives}}{\text{true negatives} + \text{false positives}}$$

and *sensitivity*, defined as:

$$\frac{\text{true positives}}{\text{true positives} + \text{false negatives}}.$$

Here true negatives and positives refer to the number of positions correctly excluded and respectively included by the filter. Correspondingly, false negatives and positive represent the counts for wrongly excluded and wrongly included positions.

In essence, specificity refers to a measure of how selective a filter or pattern is in accurately identifying relevant positions while minimising false positives. It quantifies the ability of a filter to precisely match only the desired occurrences and exclude unrelated or irrelevant ones. On the other hand, sensitivity refers to a measure of how well a filter captures and identifies all relevant positions within the reference sequences. It quantifies the ability of a filter to detect true positives while minimising false negatives. When a filter achieves 100% sensitivity, it is referred to as a *full-sensitive* or *lossless* filter.

Two commonly used filter strategies are the *pigeonhole filter* and the *counting filter*. These strategies, along with their associated index structures, form the basis of many sequence analysis applications [Holtgrewe, 2015; Rausch et al., 2008; Seiler et al., 2021; Siragusa, 2015; Weese et al., 2012]. The following subsections will briefly summarise the fundamentals of these filtration techniques.

Pigeonhole filter

Given the maximum number of errors k, the pigeonhole filter strategy involves dividing the query sequence into $k + 1$ non-overlapping segments, known as *seeds*. Each seed is then searched exactly using an index data structure. The underlying idea is that since there can be at most k errors in any k-occurrence, at least one of the $k + 1$ seeds must be identified as a true positive occurrence [Baeza-Yates and Perleberg, 1996]. This observation is commonly known as the *pigeonhole principle*.

q-gram index To efficiently filter out regions of the reference sequence that cannot contain k-occurrences, a typical implementation of the pigeonhole filter utilises a *q-gram index* built on the reference sequence x.

Definition 2.4.2 (q-gram)**.** A q-gram, often called a k-mer as well, is a sequence of the set Σ^q.

The q-gram index allows for fast retrieval of the locations where each q-gram occurs in the reference sequence, enabling efficient filtering based on the presence or absence of the seeds in the reference sequence.

The q-gram index consists of two main components: an array *pos* storing the start positions of all consecutively overlapping q-grams in x in an ascending order based on a numerical encoding and a lookup table *dir* that is used to determine the start and end positions of a particular q-gram in *pos*. The size of *pos* is $n - q + 1$, where n is the length of the reference sequence x, and q is the size of the q-grams. The *dir* array is used to determine the start and end positions of a specific q-gram within *pos*.

In its native version, each index i in the range from 0 to $|dir|$ corresponds one-to-one to a q-gram s with the i-th *lexicographic rank*. The lexicographic rank is the rank of a q-gram s within a dictionary that stores all q-grams in Σ^q sorted by their lexicographic order. This rank can be computed using the mapping function $\texttt{rank}_{\text{lex}}$, defined as:

$$\texttt{rank}_{\text{lex}}(s) = \sum_{i=0}^{|s|-1} \texttt{rank}_{\Sigma}(s_i) \cdot |\Sigma|^{|s|-1-i} \tag{2.11}$$

Here, $\texttt{rank}_{\Sigma}(s_i)$ represents the rank of the symbol s_i in the alphabet Σ (see Definition 2.2.3). The purpose of $\texttt{rank}_{\text{lex}}$ is to ensure that the numerical values of distinct q-grams do not collide. In other words, if two q-grams a and b have the same lexicographic rank, then they are equal $(a = b)$.

The pos array is filled in such a way that for all $0 \leq i < j < |pos|$, the expression $\texttt{rank}_{\text{lex}}(x_{i..i+q}) \leq \texttt{rank}_{\text{lex}}(x_{j..j+q})$ is true. This ensures that the q-gram positions in pos are sorted in ascending order according to their lexicographic ranks. If a q-gram is present multiple times in x, its start positions are stored consecutively in pos and are further ordered in ascending order.

Figure 2.8 shows an example of how a q-gram index looks like and can be used to find all 0-occurrences of a q-gram in x.

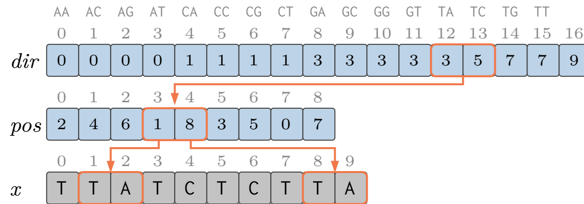


Figure 2.8: 2-gram index of $x = \texttt{TTATCTCTTA}$. To look up all text occurrences of the 2-gram TA, we first determine its hash value $\texttt{rank}_{\text{lex}}(\texttt{TA}) = 12$. The directory table dir stores at the respective positions 12 and 13 begin and end position of the interval $[3..5)$ in pos which contains the begin positions of TA in x, i.e. 1 and 8 (figure and description taken from [Weese, 2013]).

Filtering algorithm To filter with the q-gram index, we divide the query sequence y into $k + 1$ seeds, denoted by $S = \{S_0, S_1, \ldots, S_k\}$. Each seed $S_j \in S$ corresponds to the substring $y[j \cdot q..(j+1) \cdot q)$, where, without loss of generality, $q = \left\lfloor \frac{m}{k+1} \right\rfloor$.

In practice, the value of q may be limited based on the alphabet size. If q is too large, it could exceed the usable address space of a 64-bit integer. In such cases, it is possible to choose q to represent only a prefix of each seed. Now, for each seed S_j in S, we perform the following steps:

1. Compute the lexicographic rank $r = \texttt{rank}_{\text{lex}}(S_j)$.
2. Query the dir array to find the lower bound $l = dir[r]$ and the upper bound $u = dir[r+1]$ of the position interval in pos.
3. Report all start positions of S_j in x, given by the interval $pos[l..u)$.

Once a seed S_j is located using the q-gram index, the resulting locations must be verified to ensure they correspond to true k-occurrences. This verification is done by applying an online extension algorithm that aligns the prefix $y[0..j \cdot q)$ and the suffix $y[(j+1) \cdot q..m)$ to the respective regions in x. If both alignments yield an error count less than the given threshold k, the location is considered a true k-occurrence.

For the verification step, Myers' bitparallel alignment algorithm can be used with slight adaptations to the initialisation process [Weese, 2013]. This algorithm efficiently performs the alignment and allows us to determine whether a location is a valid k-occurrence based on the error count.

Counting filter

The second filter strategy is based on the *q-gram lemma* [Jokinen and Ukkonen, 1991].

Lemma 2.4.1 (q-gram lemma)**.** Let s be a k-occurrence of y in x. The q-gram lemma states that s and y must have at least $m - q + 1 - k \cdot q$ many q-grams in common.

Filters based on this lemma, such as QUASAR [Burkhardt et al., 1999], SWIFT [Rasmussen et al., 2006], or STELLAR [Kehr et al., 2011], count the number of shared q-grams between two sequences. In general, this filter strategy, partitions the reference sequence x into N overlapping segments called *bins*. Let $B = \{B_0, B_1, \ldots, B_{N-1}\}$ be the set of bins, where $b_i = x_{i \cdot b..(i+1) \cdot b}$ and, without loss of generality, $b = \lfloor \frac{n}{N} \rfloor \geq m + k$. Each bin B_i must cover at least the query sequence including possible insertions and deletions.

To apply the counting filter, the query sequence y is processed by enumerating all consecutively overlapping q-grams and counting how many of them are shared with a specific bin B_i. If the number of shared q-grams is above the threshold τ, computed by $\tau = m - q + 1 - k \cdot q \geq 1$ (see Lemma 2.4.1) for a particular bin B_i, then this bin is considered a candidate region that may contain a true k-occurrence.

However, similar to the pigeonhole filter, the regions identified by the counting filter need to be verified using an online algorithm. This verification step ensures that the potential regions indeed correspond to valid k-occurrences by aligning y to the corresponding segments in x.

Bloom filter

Instead of using a q-gram index, a counting filter can also be implemented utilising *Bloom Filters* (BFs) [Bloom, 1970]. Concretely, this approach constructs a *Bloom filter*, denoted by f_i, for each bin B_i.

A Bloom filter f consists of a bitvector represented as a sequence of the set ${\Sigma_{01}}^M$, where $\Sigma_{01} = \{0, 1\}$ denotes the binary alphabet, and, in addition, a finite set of h hash functions $H = \{H_0, H_1, \ldots, H_{h-1}\}$.

During construction, the consecutively overlapping q-grams of B_i are mapped to h positions in the range $[0..M)$ using the hash functions from H. At each of these positions, the corresponding bit in f_i is set to 1, regardless of its previous value.

To determine if a query sequence y may be present in a bin B_i, the counting process is performed. This involves iterating through the q-grams of y and checking if each q-gram is contained in the corresponding Bloom filter f_i. For each q-gram, the h positions in f_i are computed by applying all h hash functions to the q-gram. If all these positions contain a 1, the counter of B_i is incremented and finally compared with τ.

Notably, the Bloom filter is full-sensitive and will not miss any true occurrences between sequence y and x. However, it is possible to generate false positives that require unnecessary verifications using an online algorithm similar to the previous strategies. In the case of a Bloom filter f with c inserted elements, the rate of these false positives can be estimated using the formula:

$$p_{\text{fpr}}(f) = \left(1 - \left(1 - \frac{1}{M}\right)^{h \cdot c}\right)^{h} \tag{2.12}$$

Following this, we can increase the number of hash functions or enlarge the size of the bitvector in order to reduce the false positive rate.

Interleaved Bloom filter

The main issue with the described counting strategy is that querying each Bloom filter separately can become a performance bottleneck, especially when dealing with a large number of bins. However, we can address this problem by interleaving the bitvectors of the Bloom filters to form the so called *Interleaved Bloom Filter* (IBF), effectively combining the bitvectors into a single bitvector of size $M \cdot N$ [Dadi et al., 2018; Seiler et al., 2021].

In the following discussion, we assume that all Bloom filters have the same size, i.e. we have $|f_i| = M$ for all $i \in [0..N)$. To achieve this interleaving, we concatenate the corresponding bits from each of the N Bloom filters consecutively to create a new interleaved bitvector denoted by F. The first bits of all N Bloom filters are stored together in F, followed by the second bits of all filters, and so on, until all M positions are represented in F. A schematic representation of the interleaved Bloom filter can be found in Figure 2.9.

Construction To construct an IBF F, we follow a similar process as constructing a regular Bloom filter. In this case, we need to place the bits for a q-gram s in the interleaved bitvector by determining their respective positions using the h hash functions.

Since the positions obtained from the hash functions refer to the range of 0 to M, we must calculate the bin-specific offsets within the interleaved bitvector. For a specific bin B_i and hash function H_j, we multiply the resulting position $p_j = H_j(s)$ by the number of bins (N) and add i to the result. The first term, $p_j \cdot N$, determines the starting position of the interval in the F that contains the p_j-th bits of all N Bloom filters. The second term then represents the specific bit within the p_j-th Bloom filter.

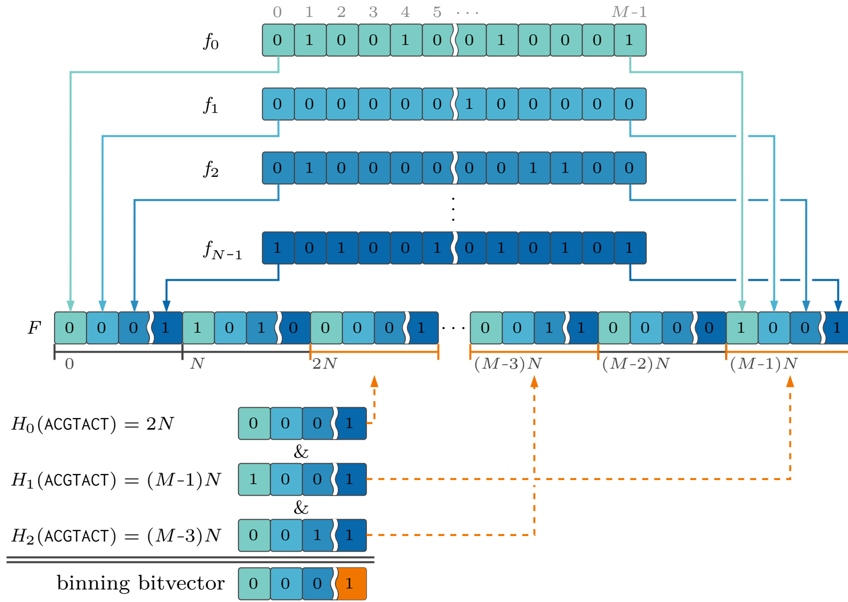


Figure 2.9: Example of an IBF F representing N differently coloured Bloom filters of length M shown at the top with three hash functions H_0, H_1, and H_2. The bitvector of the F stores the bits of the individual Bloom filters f_i in an interleaved pattern with total size $N \times M$. To determine the membership of q-gram `ACGTACT`, it is hashed with the three hash functions yielding the positions 2, $N-1$, and $N-3$. The final binning bitvector is computed from the corresponding segments of size N starting at $2 \cdot N$, $(M-1) \cdot N$, and $(M-3) \cdot N$ in the bitvector of F respectively (dashed, orange arrows). Here `ACGTACT` occurs in the last bin (Figure taken and adapted from [Dadi et al., 2018]).

Set membership query To answer a set membership query for a given q-gram s, we once again compute the positions of the q-gram using the h hash functions. Each of these hash functions produces h bit positions. Similar to the construction process, we multiply each of these positions by N to determine the starting offset of the corresponding positions of the individual Bloom filters in the bitvector of the IBF. Each computed offset indicates the start of a segment of size N. Consequently, we obtain h segments of size N for the q-gram s. To check whether s is contained in any of the bins, we reduce these segments to a single bitvector using bit-parallel AND instructions. This resulting bitvector is referred to as the *binning bitvector*.

Only if the i-th bit of the binning bitvector is 1, we increment the counter for the corresponding bin B_i. This process determines whether s is present in any of the bins. An example of a membership query can be seen in Fig. 2.9.

3 Hardware-accelerated pairwise alignment

> The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency.
>
> *(Sutter)*

In this chapter, we present and discuss various parallelisation strategies aimed at accelerating the execution of pairwise alignment algorithms on modern CPUs. Our main objective is to design and implement a versatile alignment library capable of efficiently utilising the parallelisms of current processor architectures.

The first section provides a brief explanation of the fundamental factors that can enhance the performance of software in general. We also highlight the challenges that may arise when attempting to achieve such improvements.

Next, we provide a summary of related work in the area of optimising dynamic programming algorithms for solving the pairwise alignment problem (see Section 2.3.4). This sets the stage for our approach, which involves various code-level optimisations including vectorising the DP algorithm, improving cache efficiency, and devising a scheme for dynamically scheduling and executing alignment tasks on multiple cores.

Towards the end of this chapter, we first evaluate the efficiency of our optimisations made to the DP algorithms using different use cases commonly found in bioinformatic applications and subsequently examine general aspects about our API (application programming interface) design.

3.1 Background

Since the advent of the first commodity computers, significant progress has been made in the design and manufacturing of processors, leading to continuous improvements in their performance in terms of instructions executed per second. In the early days, the number of transistors and higher clock frequencies were the primary factors driving the performance of a single processor. However, as we entered the 21st century, advancements focused on parallelism and memory structure optimisation through sophisticated microarchitecture designs (compare Fig. 3.1).

During the initial period until around 2005, there was a noticeable trend of doubling single-thread performance every 18 to 24 months. This was largely due to the doubling of transistor density that could be achieved on processors. This phenomenon came to be known as Moore's Law, named after Intel co-founder Gordon Moore, who predicted this development as early as 1965. However, this trend shifted as physical limitations prevented

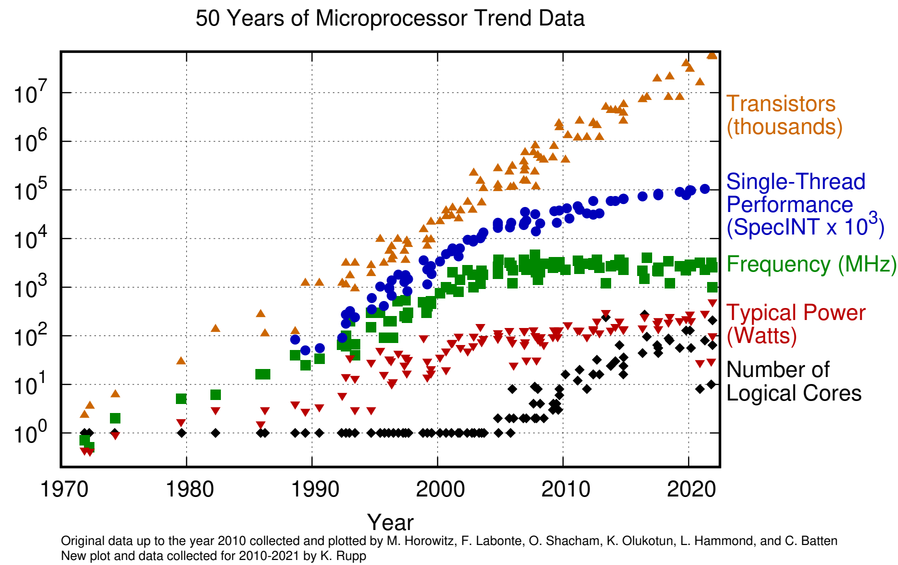


Figure 3.1: Trends in microarchitecture processor designs of the last 50 years. Figure taken from Rupp.

further increases in transistor density within a single processor. Each increase in density also exponentially increased the power requirements, caused higher heat generation as well as current leakage problems [Sutter, 2005].

To overcome these limitations, chip manufacturers adopted a different approach by distributing the number of transistors across multiple processing cores. This allowed the total number of transistors to continue doubling, but now the performance was determined by the collective workload of all processors in a system.

This shift in processor design brought about a new paradigm in software development. Herb Sutter, a prominent C++ developer, aptly captured this phenomenon in a magazine article titled 'The free lunch is over' [Sutter, 2005]. The essence of his statement was that software can no longer rely on automatic performance gains with each new generation of processors. Instead, new strategies and software designs are required to effectively distribute the workload across the available processor cores.

In modern CPUs, both commercial and high-end, various hardware techniques are employed to further enhance performance. To fully leverage the capabilities of a multiprocessor system, it is crucial to comprehend the functioning of these components. In the following sections, we will explore the four key factors of a modern CPU: the *memory hierarchy*, *data-level parallelism*, *thread-level parallelism*, and *instruction-level parallelism*. Additionally, we will explore the primary challenges that must be addressed at the code-level to optimise the utilisation of these factors. The interested reader is referred to Bakhvalov [2020]; Culler et al. [1998] and Intel [2023a] for a more detailed and comprehensive discussion of these subjects.

3.1.1 Terminologies and peformance metrics

A program consists of a sequence of instructions that are executed by the processor. These instructions encompass various operations, such as loading a value from a memory address into a register, modifying the value in a defined manner, writing it back to memory, using it as an intermediate result for subsequent instructions, and many others. The total count of instructions issued by a program to produce a result from a given input is known as the *workload*, denoted by W.

The complete set of instructions that can be executed by the processor is defined by the *instruction set architecture* (ISA). The ISA specifies how software instructions are translated into machine commands represented by sequences of 0s and 1s. Depending on the processor design and the complexity of the instructions, these commands are executed through a series of *micro-operations* (μops). A μop is the smallest operational unit that can be executed within a single clock cycle of the processor.

Throughput and Latency

When considering the performance of software or an application, we are essentially referring to the time required for its complete execution. Therefore, to enhance the performance of a program P, we need to reduce its execution time t_P, which depends on the workload W_P, and how quickly the target processor can execute these instructions.

The factors that determine the latter are the CT (cycle time), which represents the duration of a clock cycle in seconds, and the CPI (cycles per instruction), which quantifies the average number of cycles needed to complete a single instruction, also known as the *latency*. The multiplicative inverse of the CPI gives the number of IPC (instructions per cycle) executed by a processor and represents the *throughput* of a system. Note using the average number of cycles accommodates the fact that different instructions are implemented with varying numbers of μops based on the complexity of the instruction and the design of the processor architecture.

Having this, we can express the execution time as:

$$t_P = W_P \cdot \mathrm{CPI} \cdot \mathrm{CT}. \tag{3.1}$$

Therefore, it becomes clear that our primary options for improving the performance of a program lie in reducing the workload or decreasing the latency, respectively increasing the throughput.

Classification of processor architectures

In general, computer architectures can be classified according to Flynn's Taxonomy [Flynn, 1966]. Flynn's model distinguishes whether instruction streams and data streams can be processed sequentially or concurrently within a given architecture. The classes are as follows:

SISD (Single Instruction, Single Data): Architectures of this type work entirely in a serial manner, allowing only one instruction to be executed on a data element at a time. This corresponds to the early uniprocessors that had only one computing core.

SIMD (Single Instruction, Multiple Data): Architectures of this type can process multiple data elements within a single instruction simultaneously. Vector processors are well-known examples of this class.

MISD (Multiple Instruction, Single Data): In this model, different instructions are applied to a single data element at the same time.

MIMD (Multiple Instruction, Multiple Data): These architectures allow the simultaneous execution of different instructions on multiple data elements. This class includes architectures that combine several processors in a single system, such as multicore processors, where each core can independently process a separate part of the workload.

MIMD architectures can be further categorised by their memory model, distinguishing broadly between *shared memory* and *distributed memory* systems. In shared memory systems, all processors within the system are interconnected via high-bandwidth links [Mulnix, 2022], have access to the same memory and are operating under a single operating system. Typically the processors are homogeneous and have uniform access time to the memory. In this case one refers to them as SMP (shared memory multiprocessor) systems. In distributed memory systems, each processor has access to its own private memory, and they are typically connected via a local network. This also allows for hetereogenous computing by combining different processor and accelerator hardware.

Although SMP systems offer less scalability compared to distributed systems, they still play an essential role in high-performance computing, particularly as the number of independently running processors in an SMP have increased in recent years (refer to Fig. 3.1).

3.1.2 Memory hierarchy

In most CPUs, the addressable memory space is organised in a hierarchical system consisting of memory structures with varying sizes and latencies (see Fig. 3.2). This hierarchy typically includes external memory, main memory, and cache memory.

External memory provides a large and cost-effective storage space where data can be stored permanently. This includes programs we want to execute and corresponding data in the form of files. Before the CPU can execute the instructions of a program and access the associated data, they must be loaded into main memory.

Main memory, compared to external memory, is significantly faster but also more expensive, limiting its size. Despite advancements in faster memory modules, they are still considerably slower compared to the clock rate of modern processors. Consequently, loading an element from main memory into a CPU register incurs long latencies that can potentially stall the CPU.

To mitigate this latency, there is a multi-level cache memory system positioned between the CPU and main memory. The cache memories operate at the same clock frequency as the CPU registers, enabling very fast access times to the stored elements. However, cache memory manufacture is complex, expensive, and requires more energy. As a result, the

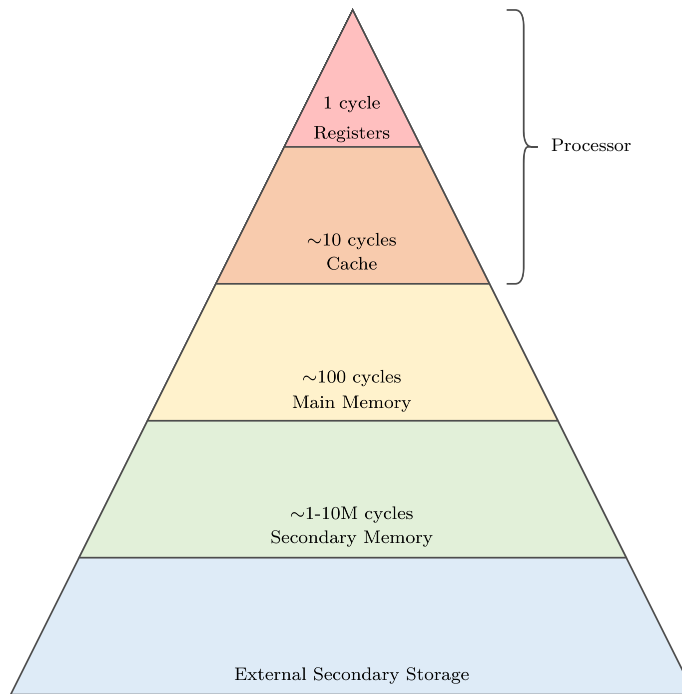


Figure 3.2: Memory hierarchy and latencies. Faster, smaller, and more expensive memory is placed closer to the processor. As the memory moves farther away from the processor, it becomes slower but offers larger memory capacity at lower costs.

cache memory is significantly smaller than the main memory. For instance the L2 cache of Intel's Xeon Scalable processor family has a size of 1 MB [Mulnix, 2022].

The memory hierarchy is utilised by the CPU as follows. Each level in the storage hierarchy represents a redundant subset of the next larger storage level. When the CPU needs to load data from a memory address in main memory, it first checks the first level of cache to determine if that address has already been loaded. If the data is found in the cache (cache hit), the corresponding element can be quickly loaded into the destination register with minimal latency.

If the data is not found in the cache (cache miss), the process is repeated with the next higher cache level. This process continues until the referenced item is either read from cache or retrieved from main memory, assuming that no unexpected exception occurs. Additionally, referenced elements are stored in all higher cache levels to expedite future access when they are needed again.

However, this means that smaller caches may need to evict another item from the cache to make room for the new item. Different conflict resolution approaches are implemented in the hardware, depending on the specific processor model. The same principles apply to writing items to the cache, although writing operations are generally more challenging to accomplish compared to reading operations. Nonetheless, writing instructions typically constitute a smaller percentage of the total number of instructions executed [Bakhvalov, 2020].

Still, understanding how caches work and how cache conflicts are resolved in hardware is crucial for improving program performance. It allows for reducing instruction latency by appropriately organising data in memory and controlling data access. The two main principles that need to be kept in mind for this are:

Temporal locality: When an element is referenced, it is likely to be referenced again in the near future. This occurs, for example, within loops or when reusing temporary values.

Spatial locality: When an element is referenced, the element with a neighbouring address is often immediately accessed. This can be observed when executing program instructions or sequentially reading elements from an array.

The design of the memory hierarchy becomes increasingly important in modern multi-core processors as the total bandwidth increases with the number of cores, surpassing the capacity of the main memory. Therefore, in addition to hierarchical memory systems, various optimisations such as multi-porting, pipelined caches, two cache levels per core, and shared third-level caches on the chip are necessary to maximise performance.

3.1.3 Data-level parallelism

Another way to optimise the performance of a program is by utilising the data-parallel *vector perocessing units* (VPUs) found in modern CPUs. These CPUs are often referred to as SIMD multiprocessors, meaning that each core is equipped with a VPU in addition to the regular processor units such as arithmetic and logic units, floating-point units, and so on.

SIMD multiprocessors greatly reduce the overall CPI by executing the same instruction on a set of independent data elements simultaneously. For example, consider the C++ code shown in Section 3.1.3, which iteratively adds the i-th element of array a to the i-th element of array b and stores the result in the corresponding position of the output array c. In scalar code, this would require executing the same instruction 16 times, incurring the associated costs for each instruction. However, by using a dedicated SIMD add instruction (e.g. `vpaddd` in the AVX512 instruction set), the same operation can be performed in a single instruction.

The corresponding assembly code, compiled with AVX512, is shown in Section 3.1.3. In the first line, the 16 data elements of a are loaded from the memory address stored in the `rdx` register into a dedicated AVX512 register (`zmm0`). These elements are then added to the 16 data elements of b referenced by the `rsi` register, and the result is stored back in `zmm0`. Finally, in line 4, the results are written back to the memory address stored in the `rdi` register.

Listing 3.1.1: Adding 4-byte packed array elements in a for-loop

```
using vec_t = std::array<int32_t, 16>;

vec_t a{}, b{}, c{};
// ... fill a, b with data ...
for (size_t i = 0; i < 16; ++i)
  c[i] = a[i] + b[i];

```

Listing 3.1.2: Assembly output for AVX512 target architecture

```
vmovdqu32  (%rdx), %zmm0
vpaddd     (%rsi), %zmm0, %zmm0
movq       %rdi, %rax
vmovdqu32  %zmm0, (%rdi)
vzeroupper
ret

```

SIMD instruction set architecture

The specific SIMD instructions available on a SIMD multiprocessor are determined by the SIMD-ISA. One of the earliest general-purpose SIMD ISAs was introduced by Intel in 1997 under the name MMX. Processors supporting MMX were equipped with eight additional 64-bit wide SIMD registers [Yu, 1997].

Over time, as newer processor generations were released, the set of SIMD instructions, as well as the width and number of SIMD registers, have been consistently expanded. For instance, contemporary high-end Intel Xeon processors are equipped with thirty-two 512-bit wide SIMD registers, and they support a range of SIMD instructions known as AVX512.

The table below (Table 3.1) provides an overview of various SIMD ISA extensions employed in Intel/AMD, ARM, and PowerPC processors.

	SSE4	**AVX2**	**AVX512**	**NEON**	**AltiVec**
Register width [bits]	128	256	512	128	128
Number of registers	8	16	32	16	32
Target architecture	x86	x86	x86	ARM	PowerPC

Table 3.1: Overview of SIMD instruction set architectures with larger register widths, number of available registers, and the target architecture they are available for.

The data elements that can be processed using SIMD instructions are typically integers (byte, word, double word, quad word) or floating-point numbers (single precision [float], double precision [double]). Each SIMD register is conceptually divided into multiple *vector lanes* based on the size of the operand. The choice of operand types has a notable influence on program performance. Figure 3.3 provides a visual representation of the widths of SIMD registers accessible in SSE4 (Streaming SIMD Extensions 4), AVX2 (Advanced Vector Extensions 2), and AVX512 (Advanced Vector Extensions 512) ISAs, as well as the number of vector lanes into which these registers can pack the operands.

Efficient VPU utilisation

In order to fully utilise the data-parallelism provided by the additional VPUs, certain aspects must be considered during programming. We will briefly explain these aspects below.

Supported operand types: The available operand types for SIMD instructions may vary depending on the SIMD ISA being used. It is important to be aware of the appropriate instruction sets and their applicability to the specific problem at hand. In some cases, it may be possible to emulate missing instructions using others, but this could increase the instruction count and negatively impact execution time. Alternatively, a different approach could lead to the use of more optimal SIMD instructions, as their CPI may differ based on the instruction and processor architecture. Complex instructions that involve permuting elements within SIMD registers can have varying latencies depending on the operand type. Similarly, their throughput may differ due to differences in available ports and execution

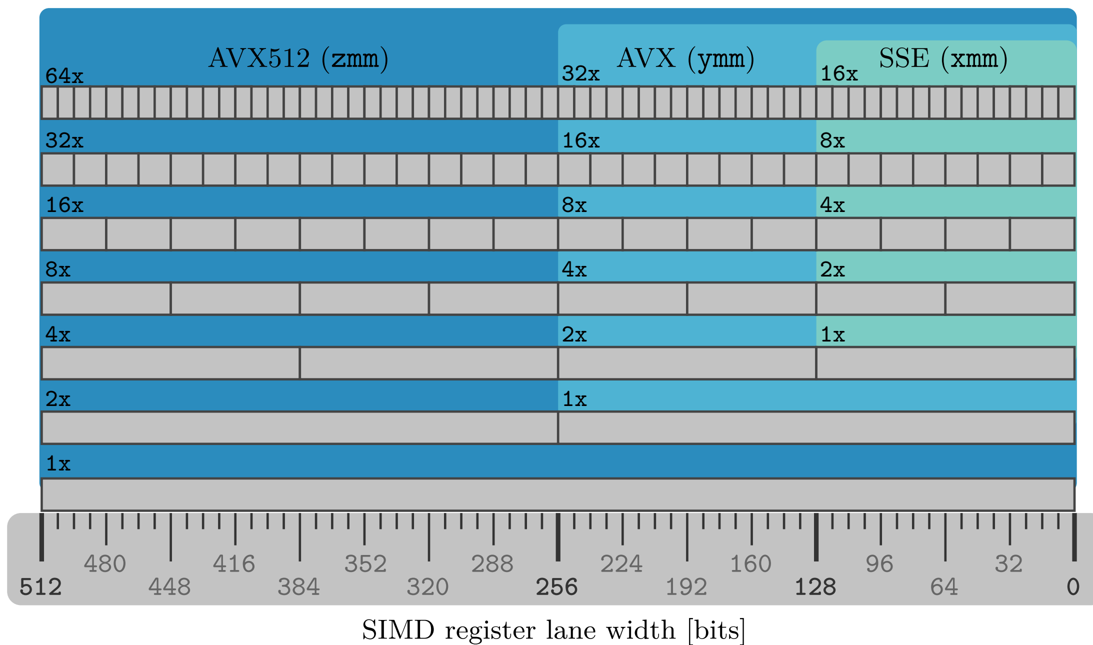
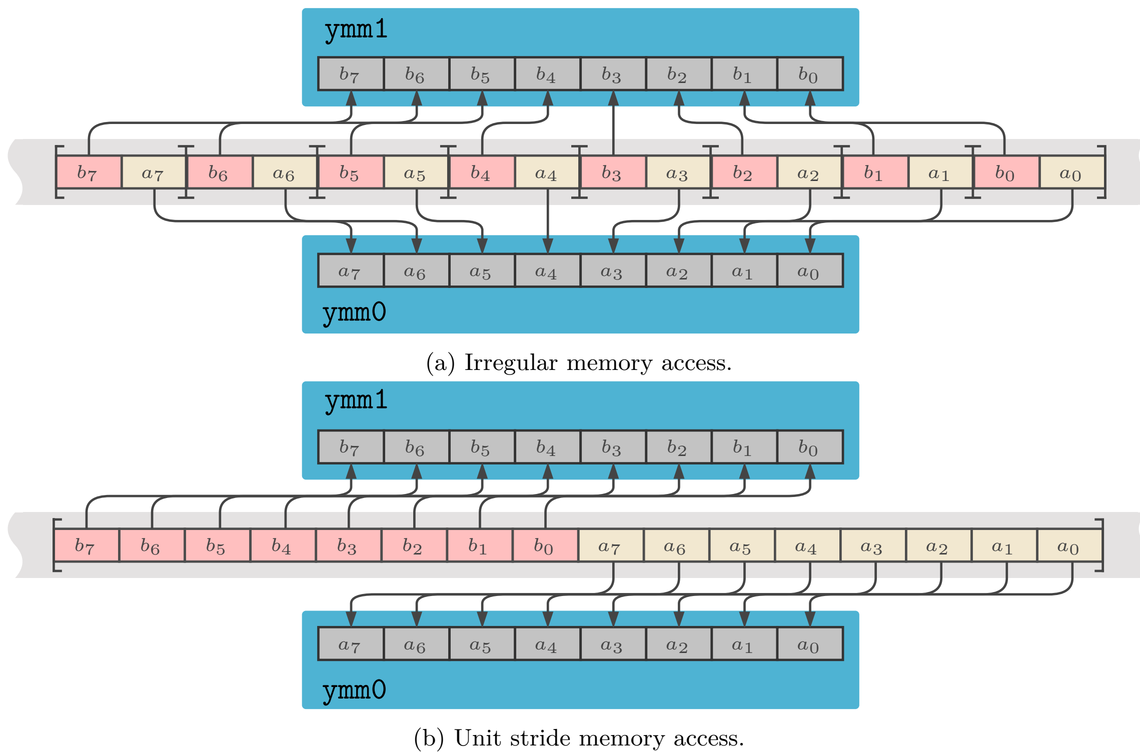


Figure 3.3: Overview of the SIMD register types and names specified by the SSE4, AVX2 and AVX512 SIMD ISA (different blue shades) and their lane counts depending on the operand sizes in bits (bottom ruler).

units for SIMD instructions, depending on the processor architecture. For example, three SIMD adds may be executed in parallel via three ports, while a shuffle operation may only be executed via a specific port. This presents further optimisation potential by effectively utilising port utilisation through suitable software design [Jeffers et al., 2016].

Data dependency: Efficient vectorisation requires ensuring that the processed operands are independent of each other, as there are no synchronisation mechanisms within the lanes of a SIMD register.

Memory access: Before data can be processed using SIMD instructions, the operands need to be loaded from memory into SIMD registers. Similarly, the results must be written back to memory after the SIMD instruction completes. Alignment and data layout play a crucial role in performance. Ideally, the data elements to be loaded are contiguous in memory and aligned to a memory address that matches the SIMD register (e.g. 16-byte aligned for SSE4, 32-byte aligned for AVX2, and 64-byte aligned for AVX512). This ensures efficient data loading into the registers, known as a unit-stride memory access pattern. However, if the data is scattered in memory, additional instructions are required to load it into the SIMD registers. Under certain circumstances, this can result in slower performance compared to the non-vectorised scalar version, if the subsequent SIMD instructions are not sufficient to compensate for the expensive gather and scatter operations. This issue can be mitigated by transforming the data layout in memory, specifically by converting from an array-of-structure (AoS) to a structure-of-array (SoA) layout. The following picture illustrates this relationship.



(a) Irregular memory access.

(b) Unit stride memory access.

Figure 3.4: Comparison of different memory access patterns to load eight yellow (right) and blue (left) fields into the 8 lanes of `ymm0` and `ymm1` respectively. In (a) the yellow and blue fields are stored in an array consisting of 8 pairs, while in (b) the fields are reordered into a single pair consisting of two arrays with 8 elements.

Auto vectorisation: As we have observed, selecting the appropriate SIMD instructions involves considering multiple factors, which significantly increases the effort required for software development. To simplify the utilisation of SIMD instructions, many compiler developers have integrated mechanisms into compilers that can automatically replace certain scalar code segments with SIMD instructions through auto vectorisation. This allows programmers to avoid writing explicitly vectorised code paths, thereby reducing the complexity of the software.

However, the capabilities of auto vectorisation are still limited, and its effectiveness heavily relies on how well the corresponding scalar code has been written to enable efficient vectorisation. For instance, small differences in coding constructs, such as using an `if` statement versus a ternary operator, can impact the effectiveness of auto vectorisation. Therefore, relying solely on this compiler feature is not always sufficient. Particularly when dealing with more complex instructions, manually optimised algorithms may be necessary. Nonetheless, by utilising SIMD libraries, such as UME::SIMD [Karpiński and McDonald, 2017], that provide a unified interface encapsulating SIMD instructions, it becomes possible to abstract the data parallelism within an algorithm and thereby reduce the code complexity.

3.1.4 Thread-level parallelism

Thread-level parallelism (TLP) provides a coarse-grained alternative to further enhance the performance of a program. It is based on the general idea to distribute work to multiple processors that can work in parallel.

Multi-threading

Typically, program execution is represented by a *process*, which manages various system resources such as program instructions, control structures for accessing operating system resources, and provides a memory address space isolated from other processes. In a multi-threaded system, a process consists of one or more *threads*, which are scheduled by the *operating system* (OS) onto available processors. Threads within the same process have shared access to the resources managed by the parent process, along with some private space to allocate *thread-local* data. The workload executed by a single thread is called a *task*.

When multiple processes run concurrently, they can be scheduled onto different processors and computed in parallel. However, this is often insufficient as individual processes may dominate the execution time. Therefore, it is desirable to split the workload of a single program into multiple tasks that can be executed concurrently on different threads.

Ideally, this allows for decreasing the execution time t_P of an *embarrassingly parallel* program P proportionally to the number of available processors. However, achieving optimal throughput for most problems involves more complex parallelisation designs. Often, concurrently running tasks exhibit data dependencies that require a sequential order of accessing memory addresses. To resolve data dependencies, additional inter-thread synchronisation primitives must be added to lock critical code paths and guarantee correct program execution.

Amdahl's law

However, such synchronisation primitives incur additional overhead, and certain sections of a program must be executed sequentially, creating areas where parallel processing is limited. This limitation also affects the scalability of a parallelised program as the number of processors increases.

Let p denote the proportion of the workload W_P of a program P that can be perfectly accelerated on an SMP with c processors. The parallel execution time is given by:

$$t_P(c) = (1 - p) \cdot t_P(1) + \frac{p}{c} \cdot t_P(1). \tag{3.2}$$

Following this, the theoretical speed up is defined as:

$$s(c) = \frac{t_P(1)}{(1 - p) \cdot t_P(1) + \frac{p}{c} \cdot t_P(1)} = \frac{1}{1 - p + \frac{p}{c}}. \tag{3.3}$$

It follows that the larger the proportion of serial execution time in the total execution time $t_P(1)$ of a parallelised program, the smaller the potential for performance improvement

through parallel processing. This observation is commonly known as Amdahl's law (see Fig. 3.5 [1]).

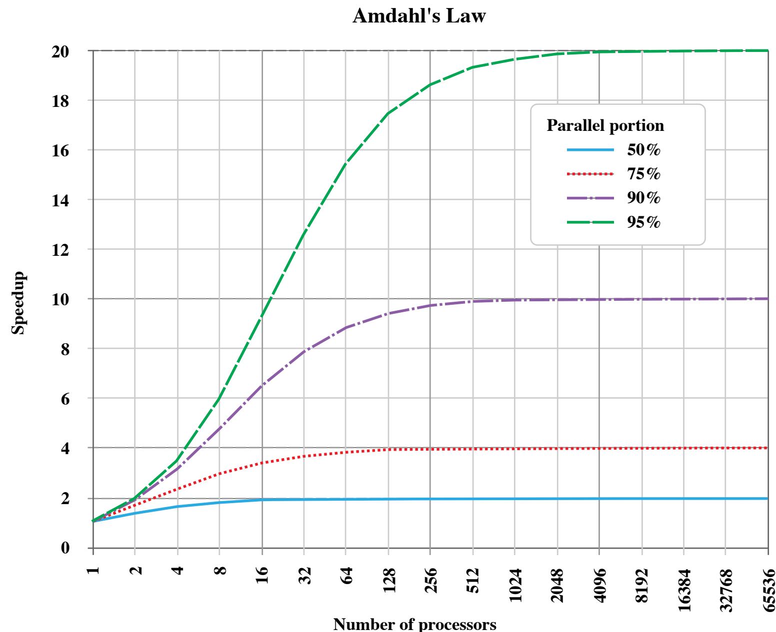


Figure 3.5: Theoretical speed up for different portions of parallelisable workloads as the number of processors increases.

Cache coherency

There are additional hardware-related considerations that one needs to be aware of when parallelising a program. Typically, each processor has its own private low-latency cache (often L1 and L2) to reduce latencies when accessing referenced elements. Therefore, as long as a thread operates on thread-local data, no other thread needs to be informed about changes made to a thread-local variable. However, when writing to a shared memory address, the hardware must ensure a consistent mapping of data with the same memory address to all threads executing in parallel on different processors. In other words, if thread 1 on processor A writes to a shared memory address that is cached by thread 2 on processor B, the corresponding cache entry of processor B must also be synchronised. This ensures that if there is a potential dependency between the executed tasks, the changes made are visible in the correct order in all participating threads.

To address this cache coherence problem, modern SMPs typically have a dedicated cache (often the L3 cache) that facilitates faster consistency and cache coherence across the

[1] Figure taken from `https://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg`; ©Daniels220 via Wikimedia Commons, CC BY-SA 3.0 `https://creativecommons.org/licenses/by-sa/3.0`; accessed 13.06.2023

processors. However, a potential issue arises as the number of processors in a shared-memory system increases, as this also increases the risk of maintaining cache consistency and coherence becoming a bottleneck

Simultaneous multi-threading

Many modern SMPs implement a feature known as *simultaneous multi-threading*, such as Intel's Hyper-Threading feature. Simultaneous multi-threading allows the sharing of hardware resources of a physical processor among two or more logical processors. From the perspective of the operating system (OS), each logical processor appears as an individual processor, enabling the scheduling of threads to run on them in parallel. This feature is particularly beneficial for applications with long latencies during execution. By interleaving the instructions of one thread with instructions from another thread sharing the same physical processor, the long latencies of one thread in the instruction pipeline can be compensated for. However, simultaneous multi-threading can also have detrimental effects, especially when both threads optimally utilise the physical resources and constantly compete for access to the execution engines.

Considering these factors, the main challenge for software developers is to decompose a program (P) into tasks that can be executed concurrently in separate threads. The goal is to distribute as much of the workload (W_P) to the available processors, while simultaneously reducing the proportion of sequentially executed code paths and minimising communication overhead as more processors are added to a multiprocessor system.

3.1.5 Instruction-level parallelism

The last aspect we would like to discuss is the exploitation of *instruction-level parallelism* (ILP) to further improve performance. Modern CPUs are typically *superpipelined* and *superscalar*, meaning they can execute more than one instruction in a cycle on average.

Superpipelined processors

Pipelined processors split an instruction into multiple stages, and each instruction must go through all stages before it retires (i.e. its execution completes). For example, in the DLX architecture, the pipeline is divided into five stages: instruction fetch, instruction decode, execution, memory access, and write back [Hennessy and Patterson, 1994].

While only one stage can be active for each instruction in a given cycle, the remaining stages are free and can be used for other instructions. For instance, after completing the instruction fetch stage, an instruction A continues with the instruction decode stage in the next cycle. At the same time, a second instruction B can enter the instruction fetch stage.

In summary, pipelined microarchitectures overlap the different execution stages of an instruction. Ideally, the pipeline can execute as many instructions in parallel as there are pipeline slots available. In the case of the DLX architecture, this would mean that, on average, a single instruction can be retired in one cycle instead of 5 cycles on a non-pipelined processor.

Superpipelined processors further subdivide the stages into smaller substages. This allows for an increase in clock speed, which is limited by the slowest stage in the pipeline. As a result, more cycles per second can be performed, and correspondingly more instructions per second can be executed, while maintaining an overall throughput of 1 IPC in an optimally utilised pipeline.

Superscalar processors

In addition, modern CPUs are equipped with multiple independent execution units in their execution engines, allowing multiple instructions to be executed in parallel. These CPUs are commonly referred to as superscalar processors. For example, Intel's Skylake microarchitecture provides 4 ALUs (arithmetic logic units). Theoretically, this allows 4 ALU instructions to retire simultaneously in each cycle (CPI=0.25, IPC=4)[2].

Execution hazards

As with the previous sections, there are hazards that limit instruction-level parallelism and hinder the optimal utilisation of the pipeline, leading to performance bottlenecks. These hazards can be categorised into structural hazards, data hazards, and control hazards. Fortunately, modern CPUs handle all pipeline hazards through hardware mechanisms. However, there are ways to mitigate the occurrence of hazards through software design, allowing further performance optimisation.

Branch misprediction A typical issue arises from the speculative execution of conditional code paths. When encountering branches in the code, the processor speculatively selects one of the two code paths and loads the corresponding instructions into the pipeline even before the result of the condition is known. If the processor predicts correctly, it can continue using the intermediate results already obtained, resulting in fast pipeline execution. However, if the branch prediction was incorrect, all intermediate results must be discarded, the active instructions are removed from the pipeline, and the alternative instructions are fetched instead. This process incurs a significant number of CPU cycles, resulting in substantial performance losses when mispredictions occur frequently.

To mitigate this, it can be beneficial to eliminate certain branches from a hot loop if they are frequently predicted incorrectly by the processor. Alternatively, branches can be replaced with conditional operations to reduce the number of branch instructions. This requires computing the results of both paths first and then propagating the result of only one of the operands to the next stage. This can be beneficial if the workloads of the two code paths are small. A typical example of this is the use of dedicated compare and blend SIMD instructions to choose elements from two vectors using an additional mask vector. By minimising branching and improving branch prediction accuracy, performance can be significantly improved.

[2] `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)#Execution_engine`; accessed 24.05.2023

Long-latency instructions Another issue that can arise is that certain instructions may have long latencies depending on the ISA and microarchitecture design. These long-latency instructions can lead to pipeline stalls where all pipelined instructions need to wait for the slowest μop to complete its stage. One possibility to improve CPU utilisation is simultaneous multi-threading, described in the previous section. Another option is for a single processor to support *out-of-order* (OOO) execution to dynamically reorder instructions. Alternatively, the compiler may be able to reorder instructions while creating the machine code of a program. In both cases, the goal is to use idle pipeline stages due to long latencies by executing other instructions instead.

The former technique is resolved entirely by the hardware, while the latter can be controlled to some extent by code-level optimisations as well. In general, compilers try to rearrange non-dependent instructions in the generated machine code. Especially inside performance-critical loops, these rearrangements can have a significant impact on pipeline throughput. Unfortunately, the optimisation opportunities are often limited by loop-carried dependencies, where some variables in the current iteration depend on the results of previous iterations. In such cases, it can be helpful to support the compiler by reorganising the execution of the loop at the software level. For instance, it may be possible to unroll the loop body and incorporate instructions from later iterations into a single block of iterations.

In practice, the options for optimising instruction-level parallelism are often limited by many other factors that need to be carefully considered. For instance, it is not always possible to extensively unroll a loop due to the potential increase in code size, which can result in cache misses in the instruction cache and adversely affect performance. Additionally, processors have a limited number of physical registers available, which may lead to register pressure and the need for register swapping. This pressure on registers can offset the benefits gained from loop unrolling. A similar effect can occur with increased pressure on instruction ports, where the number of instructions scheduled exceeds the corresponding IPC values.

3.2 Related work

With regard to alignment algorithms, various approaches and techniques have been developed in the past two decades to optimise the performance of dynamic programming (DP) algorithms. These approaches include reducing the overall workload of the DP algorithm [Marco-Sola et al., 2020; Šošić and Šikić, 2017] and changing the computation order of the DP matrix to improve parallelism utilisation. However, in the case of workload reduction, specific assumptions about the scoring scheme are often made, limiting the versatility and applicability of these solutions to specific scenarios. In the case of changing computation order, optimisations are typically tailored to specific alignment problems.

The local alignment problem has received significant attention, and numerous solutions and strategies have been proposed to enhance its performance on different processor architectures. These architectures include symmetric multiprocessors (SMPs) [Farrar, 2007; Rognes, 2011; Rognes and Seeberg, 2000], graphics processing units (GPUs) [Ahmed et al., 2019; Edans and De Melo, 2013; Khajeh-Saeed et al., 2010; Korpar and Šikić, 2013; Li et al., 2012a; Liu et al., 2013], Cell Broadband Engines [Farrar, 2008; Sarje and Aluru, 2008; Szalkowski et al., 2008], field-programmable gate arrays (FPGAs) [Li et al., 2007], as well as new accelerator technologies such as the Intel® Xeon Phi™ coprocessor [Liu and Schmidt, 2014; Liu et al.,

2014; Rucci et al., 2017]. Unfortunately, many of these approaches are integrated into specific applications and tightly coupled with the application code, making them unsuitable as stand-alone modules for other developers.

Regarding SMPs, most optimisations focused on the utilisation of SIMD instructions. A few general-purpose libraries, like SSW-library [Zhao et al., 2013] or Parasail [Daily, 2016], offer functions to facilitate SIMD instructions on such platforms. Among these, Parasail is the most elaborate library, providing implementations of the four standard DP problems: global, semi-global, overlap, and local alignment.

However, the range of applications for which pairwise alignments are computed is quite versatile, and in many scenarios, the regular versions of the DP algorithms may not be sufficient. The focus of our work lied, therefore, not only on the optimal utilisation of all performance-relevant factors of the CPU but also on a design that allows application developers to adapt and extend the existing DP algorithms by providing reusable components that can be combined with each other.

The need for such a design is justified by the fact that within the SeqAn software library itself, we counted over 30 alternative implementations of the DP algorithms that were used by numerous tools developed within our research group [Emde et al., 2012; Hauswedell et al., 2014; Holtgrewe, 2015; Kehr et al., 2011; Rausch et al., 2008; Weese, 2013] and by external developers [Roehr et al., 2017; Urgese et al., 2014].

In addition, the given input data strongly depends on the underlying problem, which has often only been rudimentarily reflected in previous solutions. Typical problems could, for example, involve the computation of many pairwise alignments between small sequences, or, in contrast to this, the computation of a few pairwise alignments of very large sequences.

After a thorough evaluation of the existing alignment algorithms implemented within SeqAn, we derived the following classification of alignment requests:

	Single alignment	**Bulk alignment**
Short sequence	SoSA (Short, Single Alignment)	SoBA (Short, Bulk Alignment)
Long sequence	LoSA (Long, Single Alignment)	LoBA (Long, Bulk Alignment)

Table 3.2: Classification of alignment requests.

With the exception of the alignment request in the top left of Table 3.2(SoSA), all requests represent a major performance bottleneck in the respective applications due to the quadratic runtime of the underlying DP algorithm. Dealing with the SoSA request is generally not a real concern as the expected execution time for aligning such small sequences is in the range of microseconds to milliseconds. Consequently, we focused on optimising the other three alignment requests.

In general, we differentiate between two main execution principles to accelerate the computation of pairwise alignments, namely *inter-sequence* execution and *intra-sequence* execution. Both principles are explained in detail below.

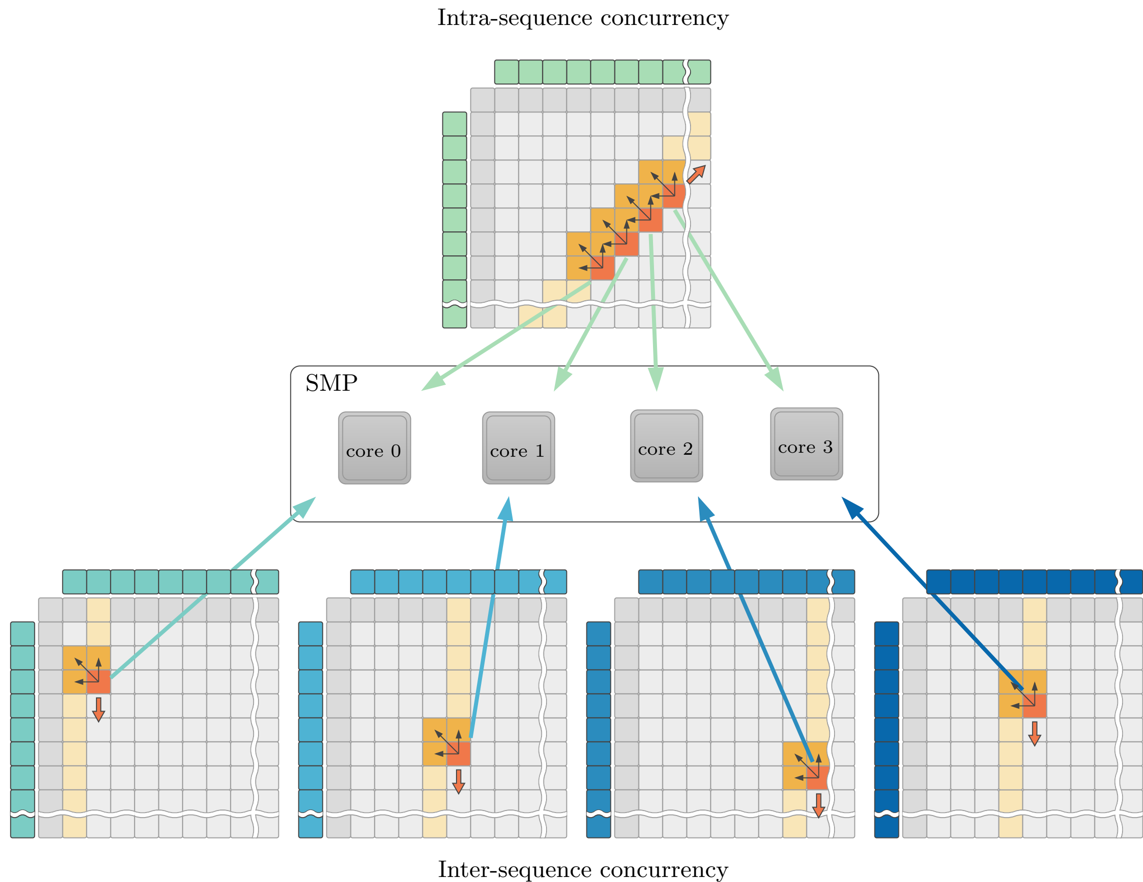


Figure 3.6: Parallel execution of five alignment instances. The green alignment instance (at the top) is parallelised using the intra-sequence execution pattern by progressing along the minor diagonals. The four alignment instances at the bottom (blue shades) are computed in parallel using the inter-sequence principle such that each DP matrix is computed independently.

3.2.1 Inter-sequence execution

The inter-sequence execution works naturally well in applications with a bulk alignment request, meaning that they require the computation of a large collection of alignments available at the same time. Since these alignments are all independent of each other, they can be easily represented by concurrent tasks that can be executed in parallel on the processors of an SMP. In the following, we refer to a particular pair of sequences for which we want to compute the optimal alignment as an *alignment instance* (AI), and the task that shall compute the alignment for a given AI as the *alignment task* (AT). Notably, this execution model lends itself to a much simpler implementation as the distribution of the concurrent ATs does not require an adaptation of the actual DP algorithm itself. This is demonstrated for the four DP matrices in Fig. 3.6.

An immediate consequence of this is that synchronisation between the executed ATs can be kept at a minimum, offering good scalability in general. In the ideal case, if the workload is homogeneous and all ATs take roughly the same time to finish, we even get an embarrassingly parallel execution scheme. Accordingly, this approach has been widely

utilised by many applications in the context of thread-level parallelisation when targeted on an SMP [Hauswedell et al., 2014; Rucci et al., 2017; Weese, 2013]. For similar reasons, some researchers have also developed an inter-sequence vectorisation strategy specifically aimed at speeding up the search of a single amino acid sequence in large databases [Alpern et al., 1995; Frielingsdorf, 2015; Rognes, 2011; Šoši, 2015].

3.2.2 Intra-sequence execution

In cases where an application can only request a single or a few alignments at a time, the inter-sequence execution approach does not scale. This is often the case in applications that compute whole-genome alignments or where the AIs are given infrequently due to more complex applications. To address this, the intra-sequence execution focuses on generating enough concurrency within a single AI. It achieves this by adapting the order in which the cells of the DP matrix are computed, allowing the dependency between cells to be removed. Specifically, the DP matrix is computed along its *minor diagonals*, which are diagonals directed from the bottom left corner towards the top right corner of the DP matrix (see Fig. 3.6). This traversal scheme enables the computation of all cells on a minor diagonal in parallel, as they have no dependency on each other.

This approach has been extensively studied, and many solutions have been proposed in the context of multi-threading for different accelerators [Edans and De Melo, 2013; Edmiston et al., 1988; Khajeh-Saeed et al., 2010; Korpar and Šikić, 2013; Li et al., 2012a; Liu et al., 2013, 2014] and vectorisation [Daily, 2016; Rucci et al., 2017; Wozniak, 1997]. Furthermore, specific strategies have been developed to optimise the intra-sequence execution time, particularly for the local alignment problem on SIMD accelerators [Farrar, 2007; Rognes and Seeberg, 2000]. These methods build on the idea that many entries in the DP matrix are 0 and can be computed speculatively along the same column in parallel. Only a fraction of these entries need to be corrected in a subsequent recursion step due to the unresolved dependency to the vertical predecessor.

Opposed to the inter-sequence execution strategy, one of the main advantages of the intra-sequence variant is that it can be used for all alignment requests in practice. Nevertheless, it has been shown that solutions based on the inter-sequence approach outperform solutions using intra-sequence execution when bulk alignment requests are available [Farrar, 2007; Khajeh-Saeed et al., 2010; Li et al., 2012a; Rognes and Seeberg, 2000]. This is because at the beginning (top left corner) and the end (bottom right corner) of the DP matrix, the number of cells on the minor diagonals is limited, which means that the parallelism offered by an SMP may not be fully utilised. Similarly, solutions based on the stripped execution pattern [Daily, 2016; Farrar, 2007, 2008; Zhao et al., 2013] require extra work to update simultaneously computed cells in the case that the speculative execution yielded a score that could influence the overall score of downstream cells.

3.2.3 Multi-level parallelism

In order to achieve optimal CPU utilisation, we need to address both SIMD-level parallelism and thread-level parallelism. For bulk alignment requests, this can be easily accomplished by combining SIMD-level parallelisation with a chunked multi-threading approach. In

this approach, the alignment instances are divided into smaller groups (chunks) and distributed to different cores of an SMP, where they can be computed using a dedicated SIMD implementation.

However, for LoSA requests, the situation becomes more complex as there are not enough alignment tasks available to fully benefit from multi-threaded execution. To overcome this, a common approach is to split the DP matrix created for two long DNA sequences into multiple tiles [Edmiston et al., 1988; Liu et al., 2014; Siriwardena and Ranasinghe, 2010] and compute these tiles along the minor diagonals in parallel. For SMPs, Liu et al. extended this approach by also applying intra-sequence vectorisation pattern within each tile, following the minor diagonals. This allows for efficient computation of each tile using both SIMD-level parallelism and intra-sequence execution.

Our contribution

In the following sections, we discuss how we have generalised the two-way execution pattern to dynamically handle various types of alignment requests while utilising the efficient inter-sequence execution pattern for data-parallel execution on the VPUs of an SMP. The foundations of this work were published in Rahn et al. [2018] in the Journal *Bioinformatics*. In this thesis, we expand upon the ideas presented in our initial work by introducing additional optimisations, particularly concerning the alignment of amino acid sequences and fully harnessing data-level and instruction-level parallelisms.

3.3 Utilising data-level parallelism

Our approach to vectorise the computation of the DP matrix is based on the inter-sequence execution pattern. To be more precise, when given a bulk of independent AIs, we do not compute the respective DP matrices one after another. Instead, we employ an interleaved execution pattern to compute the entries of the individual DP matrices. This means that initially, we compute the first entry of each DP matrix, followed by the second entry of each DP matrix, then the third entry of each DP matrix, and so on. The general idea of this approach is illustrated in Fig. 3.7.

3.3.1 Data representation

We explicitly express this interleaved execution pattern by using a dedicated *interleaved score* type, which essentially represents a fixed-size array over integers.

Definition 3.3.1 (Interleaved score)**.** Given two integers $w \in \{16, 32, 64, \dots\}$ and $p \in \{1, 2, 4, 8, \dots\}$, we write *(p,w)-vector* to denote a *packed SIMD vector* of size $\frac{w}{p}$. Correspondingly, an interleaved score, denoted by $\vec{s}$, is a (p,w)-vector over integers, i.e. $\vec{s} = (\vec{s}_i)_{i \in [0..l)}$, with $\vec{s}_i \in \mathbb{Z}$ and $l = \frac{w}{p} = |\vec{s}|$. We may also write $\langle \mathrm{c} \rangle^l$ to denote a (p,w)-vector filled with the constant c.

As described in Section 3.1.3, the size of a SIMD vector depends on the width w, which in this thesis are 16 B (SSE4), 32 B (AVX2), or 64 B (AVX512), of the SIMD register, and the size of the packed operands p, which can be 1 B (byte), 2 B (word), 4 B (double word), 8 B

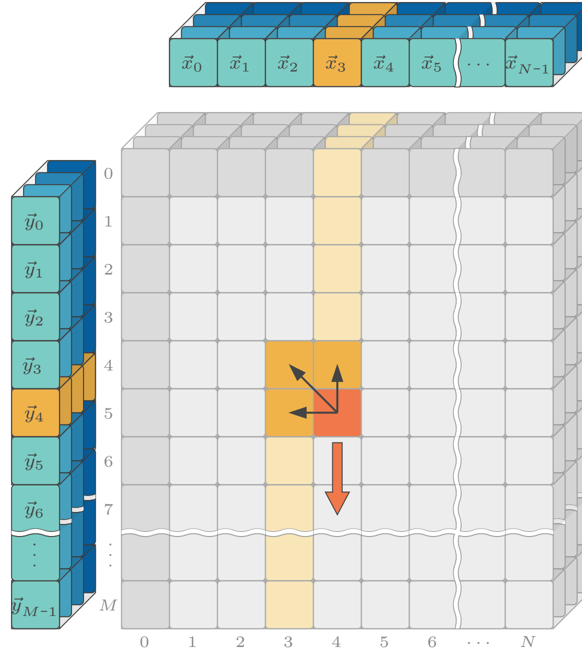


Figure 3.7: Conceptual design of the simultaneous computation of four distinct alignment instances using SIMD instructions.

(quad word), or even larger ones if there exist dedicated instructions for these sizes. Note that for conciseness reasons, we use in the following discussion the byte representation for the widths and operand sizes.

Furthermore, we assume that all unary and binary arithmetic and bit operations defined on scalar integer types, i.e. byte, word, double word, quad word, are also defined on interleaved scores, such that the respective operation is applied to each operand individually. For example, if we add two SIMD vectors $\vec{a}$ and $\vec{b}$, then

$$\vec{c} = \vec{a} + \vec{b} = \left(\vec{a}_0 + \vec{b}_0, \vec{a}_1 + \vec{b}_1, \ldots, \vec{a}_{l-1} + \vec{b}_{l-1}\right),$$

where $\vec{a}_i$ and $\vec{b}_i$ are the elements in the i-th vector lane of $\vec{a}$ and $\vec{b}$, respectively.

When comparing two interleaved scores, we obtain a new interleaved score of the same size, where each entry is either set to 0^{8p} (all bits are 0) or 1^{8p} (all bits are 1). This denotes a **false** or **true** outcome for the compared operands, respectively. In this case, we also refer to it as a *mask vector*. We can use a mask vector $\vec{m}$ to select the corresponding values from two interleaved scores using the following expression:

$$\vec{c} = \vec{m} \,?\, \vec{a} : \vec{b}, \tag{3.4}$$

such that $\vec{c}_i = \vec{a}_i$ if and only if $\vec{m}_i = 1^{8p}$ and $\vec{c}_i = \vec{b}_i$ if and only if $\vec{m}_i = 0^{8p}$. Correspondingly, the max function used inside the main recursion of the DP algorithm (see Section 2.3.4), can be expressed as $\max(\vec{a}, \vec{b}) = (\vec{a} < \vec{b}) \,?\, \vec{b} : \vec{a}$.

With these properties, we can now model the individual DP matrices corresponding to the given alignment instances as a single *interleaved DP matrix* that is defined over interleaved

score values instead of scalar ones. Specifically, given a bulk of alignment instances (X, Y), where $X = \{X_0, X_1, \ldots, X_{l-1}\}$, with $\forall i \in [0..l), X_i \in \Sigma^N$, and $Y = \{Y_0, Y_1, \ldots, Y_{l-1}\}$, with $\forall i \in [0..l), Y_i \in \Sigma^M$, we obtain an interleaved DP matrix $[\vec{\mathbf{S}}_{i,j}]^{N+1 \times M+1}$, where $\vec{\mathbf{S}}_{i,j} \in \mathbb{Z}^l$ represent a (p,w)-vector (see Fig. 3.7).

Note that for simplicity, we assume without loss of generality that $l = \frac{w}{p} = |\vec{\mathbf{S}}_{i,j}|$. Importantly, this interleaved DP matrix can be computed using the same DP algorithm as a scalar DP matrix, as shown in Fig. 2.6 in the preliminary section (see Section 2.3.4). The score of the individual optimal alignments can then be located equivalently. Moreover, this representation natively supports the computation of alignments with affine gaps, tracing the path of the optimal alignments, and even computing the DP algorithm with a band.

However, this approach only works seamlessly if the sequences in X have the same size and the sequences in Y have the same size, as is often the case for short-read sequencing data, for example. We use the term *homogeneous sequence collections* to denote this property of a sequence collection. On the other hand, if the sizes of the sequences in a sequence collection differ, we refer to it as a *heterogeneous sequence collection*. We will first focus on homogeneous sequence collections to describe the main ideas and fundamental operations needed to speed up the alignment computation using SIMD instructions.

3.3.2 Memory transformation

The first step before computing the interleaved DP matrix involves a linear transformation to convert both sequence collections, X and Y, into arrays of (p,w)-vectors, obtaining their interleaved representation. The interleaved sequences are denoted by $\vec{x}$ having a size of N and $\vec{y}$ having a size of M, respectively. Moreover, we store the ranks of the symbols of the original sequences in the interleaved memory representation. This initial transformation step is necessary to avoid expensive gather instructions that exhibit long latencies to load the symbols into the SIMD registers during each iteration of the DP algorithm (see Section 3.1.3).

Algorithm 3.1: `AoS2SoA`

Input: X, w, p

```
1  l ← |X|
2  N ← 0
3  for i ← 0 to l do
4      N ← max(N, |X_i|)
5  x⃗ ← new array over (p,w)-vectors of size N
6  for i ← 0 to N do
7      for j ← 0 to l do
8          if i < |X_j| then
9              x⃗_i[j] ← X_j[i]
10         else
11             x⃗_i[j] ← $
12 return x⃗
```

The scalar algorithm `AoS2SoA` used to transform the input sequence collections is shown in Algorithm 3.1. First, the maximum sequence size, N, is determined by performing a linear scan over the sequence collection (Lines 3 and 4). Next, a new array, $\vec{x}$, is allocated, which will contain N (p,w)-vectors. The array is then filled by executing the nested for-loop. The outer for-loop iterates over the sequences of X, while the inner for-loop iterates over the lanes of the i-th (p,w)-vector in $\vec{x}$.

During each iteration, if the j-th sequence in X has a symbol at position i, that symbol is assigned to the corresponding lane of $\vec{x}_i$. Otherwise, a special padding symbol, denoted as \$ (where $\$ \notin \Sigma$), is assigned to that lane. We will discuss the usage of the padding symbol later in Section Section 3.3.3 when handling hetereogenous sequence collections. Finally, the transformed sequence, $\vec{x}$, is returned.

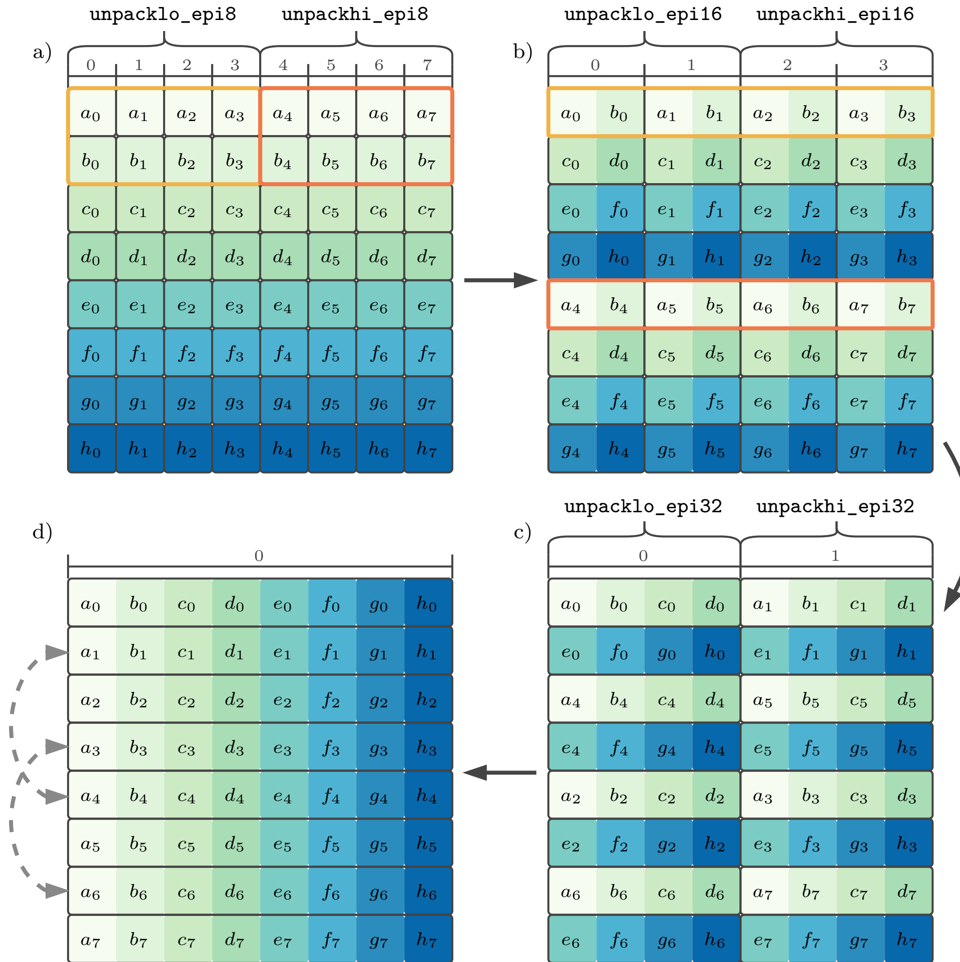


Figure 3.8: Transposing eight consecutive symbols of eight sequences (a, b, c, d, e, f, g, h) using dedicated `unpack_lo` and `unpack_hi` SIMD instructions in three steps: a) unpacking 1-byte lanes, b) unpacking 2-byte lanes, c) unpacking 4-byte lanes. The low bytes are put to the upper half of the matrix (yellow box) and the high bytes are put to the lower half of the matrix (orange box). In the final step d) the second row is swapped with the fifth row and the fourth row is swapped with the seventh row to obtain the correctly transposed matrix.

Vectorised transformation

To speed up the initial transformation we implemented a vectorised version of `AoS2SoA` using a series of SIMD *unpack* instructions for the special case where the sizes of the packed operands is one, i.e. $p = 1$. Figure 3.8 illustrates this procedure using an 8×8 matrix.

The general procedure for transposing an $l \times l$ matrix using SIMD instructions involves repeatedly unpacking the low and high lanes of two adjacent rows in the matrix. The procedure starts with the smallest operand size and, in each step, unpacks the lanes corresponding to the next larger operand size (i.e. 1, 2, 4, 8, 16, etc.). The unpacked low lanes are stored in the upper half, while the unpacked high lanes are stored in the lower half of an intermediate matrix. This ensures that in the last unpacking step, all rows of the intermediate matrix have the correct order.

However, since some rows are now placed at wrong row indices, certain rows need to be swapped. In Fig. 3.8, this is illustrated by the second and fifth row and the fourth and seventh row. The final order can be precomputed, allowing us to directly assign the rows to their correct positions in the last step.

Overall, this procedure requires only $\mathcal{O}(l \log_2 l)$ read and write instructions opposed to the $\mathcal{O}(2l^2 - 2l)$ instructions required by the scalar version. In terms of numbers, we can reduce the workload for the transpose operation for different SIMD ISAs, as shown in Table 3.3.

	SSE4 ($l = 16$)	**AVX2** ($l = 32$)	**AVX512** ($l = 64$)
	instruction count		
scalar	480	1984	8064
SIMD	96	224	512
	Throughput [GB/s]		
scalar	1.09	1.69	2.07
SIMD	1.14	3.95	8.85

Table 3.3: The first part gives the numbers of read and write instructions needed to transpose a $l \times l$ matrix using element-wise transpose and the vectorised version for the respective SIMD ISA. The second part gives the total throughput in GB/s of transposing $\frac{4\,096\,000\,\mathrm{B}}{l^2}$ many $l \times l$ matrices.

3.3.3 Vectorised scoring schemes

Having established the general idea of computing the interleaved DP matrix, we now focus on the specific adaptations we made to vectorise the scoring schemes. The gap score functions were trivially adapted by filling the parameters g_{ext} and g_{opn} into each lane of the interleaved scores. In the following, we will use $\vec{g}_{\text{ext}} = \langle g_{\text{ext}} \rangle^l$ and $\vec{g}_{\text{opn}} = \langle g_{\text{opn}} \rangle^l$ to refer to the interleaved gap scores.

Considering only homogeneous sequence collections, we did the same for the unitary substitution score function. In this case, we filled the score values for a match and mismatch into the corresponding interleaved score, i.e. $\vec{m}_{=} = \langle m_{=} \rangle^l$ and $\vec{m}_{\neq} = \langle m_{\neq} \rangle^l$. Then, we

used $(\vec{x}_{i-1} = \vec{y}_{j-1})\,?\,\vec{m}_{=} : \vec{m}_{\neq}$ to compute the interleaved substitution score for the current coordinate (i, j).

Unfortunately, the same trivial adaptation could not be done for the matrix substitution score function. Instead, we stored the scalar values of the substitution matrix in a linearised array and performed a gather operation to load the respective scalar score values from this array. In subsequent sections, we will explain how we improved the performance of this expensive gather operation by utilising more efficient shuffle instructions.

3.3.4 Processing heterogeneous sequence collections

To generalise this approach to heterogeneous sequence collections where the sequence lengths differ, we used two masking vectors $\vec{m}_{\text{x}}$ and $\vec{m}_{\text{y}}$ of size $N + 1$ and $M + 1$, respectively, to track the different ends of the respective sequences. Here, we assume that $N = \max(n_0, n_1, \ldots, n_{l-1})$, with $n_k = |X_k|$, and $M = \max(m_0, m_1, \ldots, m_{l-1})$, with $m_k = |Y_k|$, each representing the size of the largest sequence within the respective collection.

In each iteration, we checked if the currently processed coordinate (i, j) required an alignment-specific post-processing step based on the values in the two masking vectors at the corresponding positions, i.e. $\vec{m}_{\text{x}}[i]$ and $\vec{m}_{\text{y}}[j]$, and the specific alignment algorithm used. In the case of computing a global alignment, for example, the post-processing operation was only issued when $\vec{m}_{\text{x}}[i] \,\&\, \vec{m}_{\text{y}}[j] \neq \langle 0^{8p} \rangle^l$ and only the result of the lanes that compared equal to $\langle 1^{8p} \rangle^l$ were included.

The advantage of this separate masking approach was that we could utilise all of the existing scalar DP implementation by merely wrapping the corresponding functions. However, this masking led to a strong reduction in the overall performance gain compared to the unmasked version. Hence, we extended the initial sequence transformation to check if the given sequence collections were homogeneous and used the more efficient version if this was the case.

Optimised vectorisation for hetereogenous sequence collections

We recently improved the performance of the heterogeneous computation by replacing the expensive masking with a more efficient strategy. In this approach, we extended smaller sequences in X and Y with cleverly selected padding symbols that are not contained in the underlying sequence alphabet. The general idea is to populate the final score values of the individual DP matrices to the last row and column of the interleaved DP matrix along their diagonals (see Fig. 3.9). By doing so, we could infer the correct score values of each individual DP matrix from the last row and column, rather than interrupting the computation of the DP matrix to run sequential operations.

Local alignments In the case of computing a local alignment, we used two distinct *padding symbols*, namely $\$_{\text{x}}$ and $\$_{\text{y}}$, to pad smaller sequences in X and Y, respectively. Formally, we have that $\$_{\text{x}}, \$_{\text{y}} \notin \Sigma$, and $\$_{\text{x}} \neq \$_{\text{y}}$. As a consequence, substituting a symbol with either padding symbol yields a mismatch.

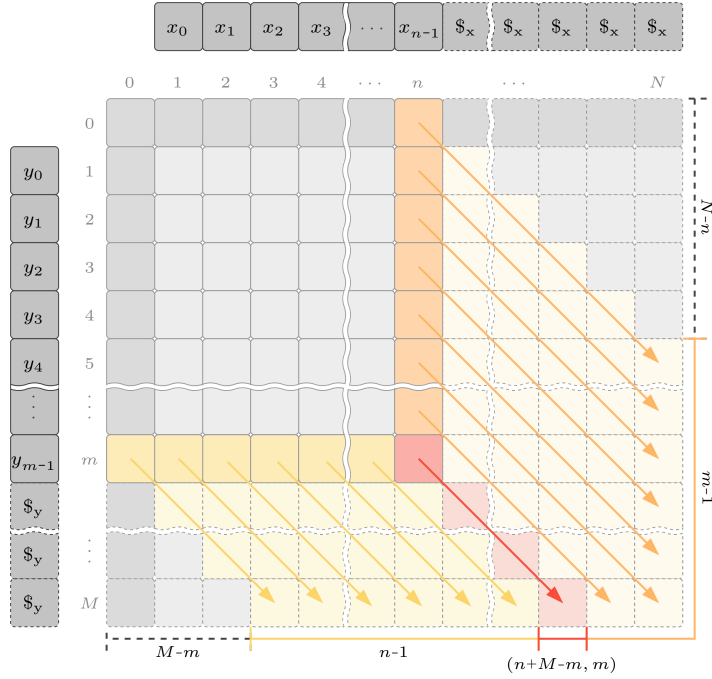


Figure 3.9: Scores of the last column and row of the alignment instance (x, y) are populated along their diagonals to the last column and row of the interleaved DP matrix, as indicated by the arrows. The start of the projected cells in the last row (yellow tiles) and column (orange tiles) are shifted by $N - n$ and $M - m$ respectively (dashed crossbars). The bottom right corner (n, m) (red tiles) is projected to $(n + M - m, m)$.

Score values computed for entries beyond the dimensions of the original DP matrix will only decrease, and thus must be lower than the largest score computed within the original dimensions. In the case of using a substitution score matrix, it suffices to use a single padding symbol and set the corresponding row and column entries to the lowest score value present in the entire matrix, e.g. -4 in the case of a BLOSUM62 matrix.

Global alignments In the other cases, where a global alignment, semi-global alignment, or overlap alignment is computed, we populate the score values of the last row $\mathbf{S}[*, m_k]$ and column $(\mathbf{S}[n_k, *])$ of the k-th DP matrix in such a way that, after reaching the original ends, the score values of the last column and row are increased only by the scores for receiving a match.

Knowing the dimensions of the individual DP matrices, i.e. (n_k, m_k) for the k-th alignment instance (X_k, Y_k), and the dimensions of the interleaved DP matrix, i.e. (N, M), we can infer the original scores of the last row and column of an individual DP matrix by subtracting the length of the padded suffix in the respective dimension multiplied by the score for a match. For example, in Fig. 3.9, we can infer the original scores corresponding to the last row m (yellow tiles) by subtracting $(M - m)\vec{m}_{=}$ from each value in the range $\{M - m, \ldots, n + M - m\}$ over the last row of the interleaved DP matrix, for some row m and column n.

Similar to the local alignment using a matrix score function, we extended the substitution score matrix with an additional row and column for the padding symbol \$, but we set the

corresponding values to the largest positive value, e.g. 11 in the case of a BLOSUM62 matrix, rather than the lowest value.

For the unitary score function, however, we used a less trivial solution to avoid an additional comparison for the currently observed symbol pair $\vec{x}_{i-1}$ and $\vec{y}_{j-1}$. Our solution requires only one additional padding symbol \$, which is used to pad shorter sequences in both sequence collections. We initialised it by setting the most significant bit of the underlying operand type, i.e. $\$ = 10^{8p-1}$.

Having this, we replaced the equality comparison of the vectorised unitary score function with:

$$((\vec{x}_{i-1} \,\widehat{}\, \vec{y}_{j-1}) \leq \langle 0 \rangle^l) \,?\, \vec{m}_= : \vec{m}_{\neq}.$$

In this expression, we compute the bitwise XOR ($\widehat{}$) between both symbols, and whenever the result, interpreted as a signed integer, is negative or 0, we return the scores for a match for this particular lane. The result of the bitwise XOR can only be 0 if the compared symbols are identical. If only one of the compared symbols is the padding symbol, the sign bit in the resulting operand will be set, yielding a negative value. In all other cases, where both compared symbols differ and neither of them is the padding symbol, the resulting operand must have a value strictly greater than 0.

Theoretically, this approach limits the applicable alphabet type of the sequences. For the adaptation of the global alignment, we require that the original alphabet size $|\Sigma|$ is strictly less than 2 to the power of $8p - 1$, so that the most significant bit is not occupied. Similarly, for the local alignment, the alphabet size must be less than $8p - 2$.

However, for the majority of relevant applications, we can assume an alphabet that can be modelled with the 7-bit ASCII character set, which allows us to use the smallest operand size of one byte. This covers a wide range of commonly used characters and symbols.

Offset correction In cases where the main diagonal of an individual DP matrix is shifted to the left, i.e. $m_k - M < n_k - N$, or right, i.e. $n_k - N < m_k - M$, with respect to the main diagonal of the interleaved DP matrix, a part of the column coordinates or row coordinates are projected to the last row or column, respectively. For example, the coordinate $(n, m - 1)$ in Fig. 3.9 is projected to the last row of the interleaved DP matrix rather than the last column (last orange diagonal above the red diagonal), because in this example $m_k - M < n_k - N$. To account for this, the affected score values are corrected by subtracting o many match scores, where o is a position-specific offset representing the distance between the column index of the projected last row coordinate (in this example $n + M - m + 1$) of the interleaved DP matrix and N or, if the main diagonal is shifted towards the other side, the distance between the row index of the projected last column coordinate of the interleaved DP matrix and M.

Improved vectorisation of matrix substitution scores

As discussed earlier, we encountered performance issues with the gather instruction used to load score values from a substitution score matrix, particularly due to its high latency and limited availability for certain operand types [Intel, 2023b]. In cases where the gather instruction is not available, we resort to a sequential lookup as a fallback, further diminishing

the expected performance gains. To address these issues, we have recently implemented two strategies to improve the performance of aligning amino acid sequences.

Profile substitution scores As the first strategy we implemented a column-wise *profile score function*, where a single sequence is aligned to a bulk of sequences. This scenario commonly arises when searching for a query sequence in a protein database [Rognes, 2011].

In this approach, we pre-compute at the beginning of each column of the interleaved DP matrix a column-specific lookup table, denoted as $\vec{P} = \left(\vec{P}_0, \vec{P}_1, \ldots, \vec{P}_{|\Sigma|-1}\right)$. The lookup table contains elements represented as $(p{,}w)$-vector, which store the scores for aligning symbols from Σ with the current interleaved symbol $\vec{x}_{i-1}$ at column i. The profile for column i can be constructed efficiently using a series of shuffle, arithmetic, and logic SIMD instructions, with the computational complexity linear to the size of the alphabet [Rognes, 2011]. With the constructed profile, we obtain the substitution scores in the inner loop of the DP algorithm by simply accessing the cached interleaved scores using $\vec{P}[\texttt{rank}_\Sigma(y_{j-1})]$ at each position j in column i.

Although the profile score function provides much faster run times compared to the previously discussed method using a gather instruction, it is only applicable to specific applications that involve aligning a single sequence to a bulk of sequences. Therefore, it cannot be used as a general replacement for the gather instruction.

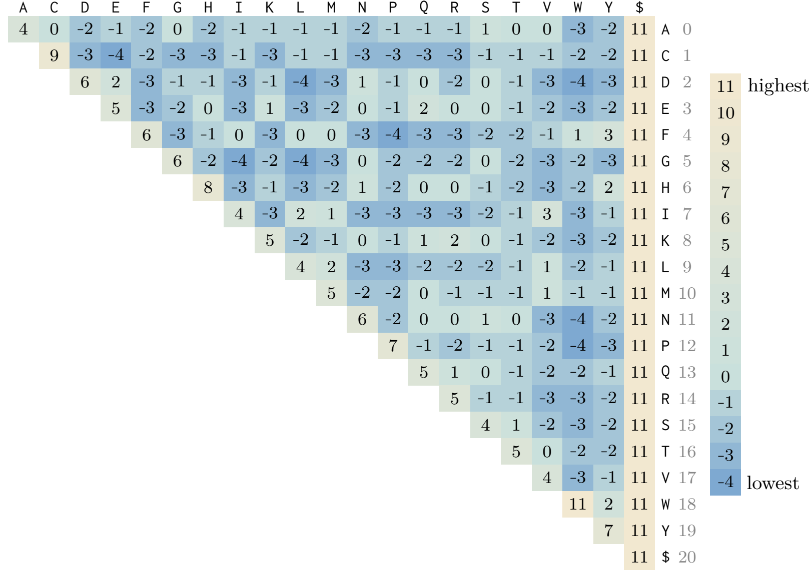
Optimising matrix gather To address this limitation, we have developed a new approach that replaces the gather instruction with a series of shuffle instructions exhibiting a lower CPI than gather instructions. Note that at the point of writing this thesis, we have only implemented it for the AVX512VBMI and AVX512BW ISA using byte operands. More specifically, we used a cross lane permutation instruction (`vpermi2b`) to select from two source $(1, 64)$-vectors values of the substitution score matrix. On Intel's Icelake server CPUs the CPI of this instruction is 4, whereas the CPI of `vpgatherdd` (gather of double word integral types) is 8 [Abel and Reineke, 2019][3]. Since gather instructions are only available for 4-byte and 8-byte operand types, we need to wrap four `vpgatherdd` invocations in a series of unpack and pack instructions to emulate a single invocation of `vpermi2b`, such that the accumulated latency increases even further.

Technically, the `vpermi2b` instruction allows us to select 128 operands from two $(1, 64)$-vectors. By combining two calls to this instruction, we are able to cover the maximum addressable index range of 1-byte operand types, spanning from 0 to 255. Unfortunately, this index range is too small to store the entire substitution matrix for amino acid alphabets, denoted by Σ_{AA}, provided that $|\Sigma_{\mathrm{AA}}| = 20$. However, we can make use of the symmetry of most standard amino acid substitution score matrices, such as the BLOSUM62 matrix (compare Fig. 2.2). Particularly, we only need to store the upper triangular half of the matrix, including the main diagonal as demonstrated in Fig. 3.10.

The size of this triangular matrix is given by:

$$\sum_{i=0}^{|\Sigma|} i = \frac{|\Sigma|(|\Sigma| + 1)}{2}.$$

[3]CPI data are taken from `https://uops.info/table.html`



A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y	$		
4	0	-2	-1	-2	0	-2	-1	-1	-1	-1	-2	-1	-1	-1	1	0	0	-3	-2	11	A	0
	9	-3	-4	-2	-3	-3	-1	-3	-1	-1	-3	-3	-3	-3	-1	-1	-1	-2	-2	11	C	1
		6	2	-3	-1	-1	-3	-1	-4	-3	1	-1	0	-2	0	-1	-3	-4	-3	11	D	2
			5	-3	-2	0	-3	1	-3	-2	0	-1	2	0	0	-1	-2	-3	-2	11	E	3
				6	-3	-1	0	-3	0	0	-3	-4	-3	-3	-2	-2	-1	1	3	11	F	4
					6	-2	-4	-2	-4	-3	0	-2	-2	-2	0	-2	-3	-2	-3	11	G	5
						8	-3	-1	-3	-2	1	-2	0	0	-1	-2	-3	-2	2	11	H	6
							4	-3	2	1	-3	-3	-3	-3	-2	-1	3	-3	-1	11	I	7
								5	-2	-1	0	-1	1	2	0	-1	-2	-3	-2	11	K	8
									4	2	-3	-3	-2	-2	-2	-1	1	-2	-1	11	L	9
										5	-2	-2	0	-1	-1	-1	1	-1	-1	11	M	10
											6	-2	0	0	1	0	-3	-4	-2	11	N	11
												7	-1	-2	-1	-1	-2	-4	-3	11	P	12
													5	1	0	-1	-2	-2	-1	11	Q	13
														5	-1	-1	-3	-3	-2	11	R	14
															4	1	-2	-3	-2	11	S	15
																5	0	-2	-2	11	T	16
																	4	-3	-1	11	V	17
																		11	2	11	W	18
																			7	11	Y	19
																				11	$	20

Figure 3.10: Upper triangular half of the BLOSUM62 substitution score matrix.

This means, we can encode any substitution score matrix with an alphabet size of up to 22, which satisfies $\frac{22 \cdot 23}{2} = 253 \leq 256$. The corresponding score values are then stored in a linear fashion in four consecutive $(1, 64)$-vectors. This allows us to efficiently access the required score values using the `vpermi2b` instructions, despite the limited index range.

In order to implement this strategy, we needed to compute the selection index that corresponds to the linearised index of the triangular matrix for a given pair of interleaved symbols $\vec{a} = \vec{x}_i$ and $\vec{b} = \vec{y}_j$. Let $\vec{\alpha} = \langle|\Sigma|\rangle^l$, then this can be done with the formula:

$$\frac{\vec{\alpha}(\vec{\alpha} - \langle 1 \rangle^l)}{\langle 2 \rangle^l} - \frac{(\vec{\alpha} - \min(\vec{a}^{\mathrm{r}}, \vec{b}^{\mathrm{r}}))(\vec{\alpha} - \min(\vec{a}^{\mathrm{r}}, \vec{b}^{\mathrm{r}}) - \langle 1 \rangle^l)}{\langle 2 \rangle^l} + \max(\vec{a}^{\mathrm{r}}, \vec{b}^{\mathrm{r}}), \tag{3.5}$$

where $\vec{a}^{\mathrm{r}} = (\texttt{rank}_\Sigma(\vec{a}_0), \texttt{rank}_\Sigma(\vec{a}_1), \ldots, \texttt{rank}_\Sigma(\vec{a}_{l-1}))$ and, analogously the same for $\vec{b}^{\mathrm{r}}$. Note this formula works for a general upper triangular matrix $\mathbf{T}$ as long as the row index is greater or equal to the column index since $\mathbf{T}[i, j] = \mathbf{T}[j, i]$. As such, we use the minimum of both ranks to select the column index and use the maximum of both ranks as the row index.

However, there are two main issues with this method. First, it involves a multiplication and a division which are not supported for `byte` operand types [Intel, 2023b]. Instead, the compiler adds extra instructions to convert the $(1, 64)$-vector to two $(2, 64)$-vectors to perform the operations and converts the intermediate results back to a $(1, 64)$-vectors. Second, the multiplication produces arithmetic overflows on 1-byte operands distorting the computed index.

To address the first issue, we precomputed the starting offsets of each symbol in the linearised triangular matrix during the initial preprocessing of the sequence collections, such that the total overhead is only linear rather than quadratic. For the latter problem, we used an additional mask vector $\vec{m}$ that marks all lanes of a (p,w)-vector $\vec{a}^{\mathrm{r}}$ containing an uneven rank

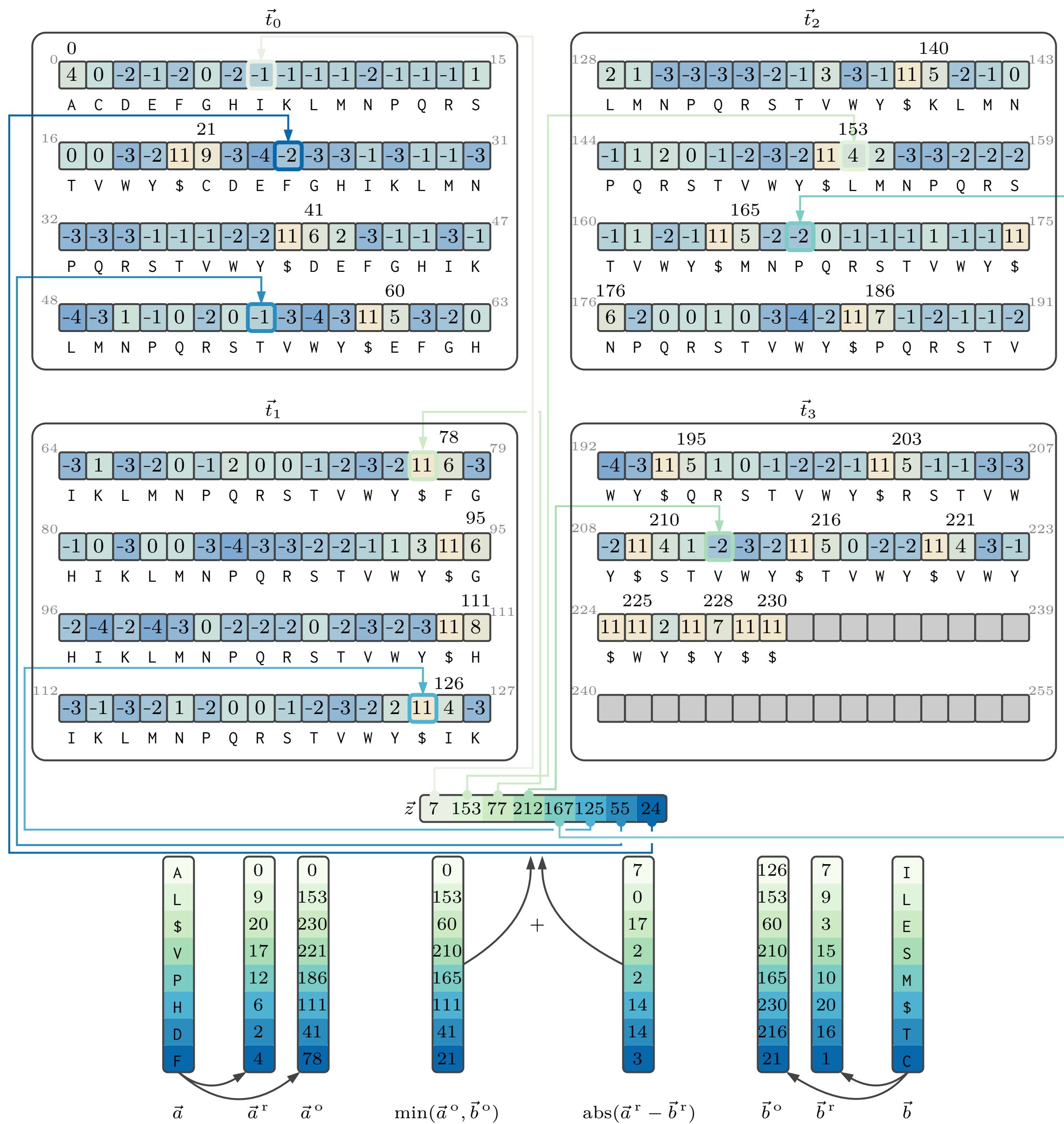


Figure 3.11: Illustration of the selection process for two interleaved symbols $\vec{a}$ and $\vec{b}$ with 8 lanes each. The original symbols are converted to a pair consisting of their alphabet rank ($\vec{a}^{\mathrm{r}}$, $\vec{b}^{\mathrm{r}}$) and their offset in the linearised triangular substitution score matrix ($\vec{a}^{\mathrm{o}}$, $\vec{b}^{\mathrm{o}}$). By taking the minimum of both offsets and adding to them the absolute values of the differences between the ranks, the final selection index $\vec{z}$ is computed. This is used to retrieve the corresponding score values of the BLOSUM62 score matrix that is encoded by four $(1, 64)$-vectors. Each of these vectors consists of four 16-byte lanes highlighted with the left and right selection index in light gray. The black indices above the entries mark the start offset of the respective amino acid in the linearised triangular substitution matrix. The final interleaved score vector contains $(-1, 4, 11, -2, -2, 11, -1, -2)$.

and cached the result of the first term of Eq. (3.5) in a temporary variable $\vec{c} = \langle \frac{|\Sigma|(|\Sigma|-1)}{2} \rangle^l$. Using this, we can now compute the offset for a symbol $\vec{a}$ with:

$$\begin{aligned}
\vec{u} &= \vec{a}^{\mathrm{r}} \,\&\langle 1\rangle^l \\
\vec{o} &= \left(\vec{u}\,?(\vec{\alpha} - \vec{a}^{\mathrm{r}} - \langle 1\rangle^l) : (\vec{\alpha} - \vec{a}^{\mathrm{r}}) \gg \langle 1\rangle^l\right) \cdot \left(\vec{u}\,?(\vec{\alpha} - \vec{a}^{\mathrm{r}}) : (\vec{\alpha} - \vec{a}^{\mathrm{r}} - \langle 1\rangle^l)\right) \\
\vec{a}^{\mathrm{o}} &= \vec{c} - \vec{o}
\end{aligned}$$

replacing on the one hand the division by 2 with faster bit shifts and on the other hand avoiding arithmetic overflows by first dividing the even ranks and then multiplying the corresponding uneven rank to the intermediate results. After the initialisation we replace each input $(1, 64)$-vector with an ordered pair consisting of the ranks of the symbols, and, as the second value, the precomputed start offsets of the symbols in the linearised triangular score matrix.

Let $(\vec{x}^{\mathrm{r}}_{i-1}, \vec{x}^{\mathrm{o}}_{i-1})$ and $(\vec{y}^{\mathrm{r}}_{j-1}, \vec{y}^{\mathrm{o}}_{j-1})$ represent the interleaved rank and offset values of the i-th column and the j-th row, respectively. Furthermore, let $\vec{t} = \left(\vec{t}_0, \vec{t}_1, \vec{t}_2, \vec{t}_3\right)$ be the fixed-size array storing the linearised triangular matrix in four $(1, 64)$-vectors. We can now replace the expensive gather instruction in the inner-loop of the DP algorithm with

$$\vec{z} = \min(\vec{x}^{\mathrm{o}}_{i-1}, \vec{y}^{\mathrm{o}}_{j-1}) + \mathrm{abs}(\vec{x}^{\mathrm{r}}_{i-1} - \vec{y}^{\mathrm{r}}_{j-1}) \quad (3.6)$$

$$\vec{s} = (\vec{z} < \langle 128\rangle^{64})\,?\,\texttt{vpermi2b}(\vec{t}_0, \vec{z}, \vec{t}_1) : \texttt{vpermi2b}(\vec{t}_2, \vec{z}, \vec{t}_3). \quad (3.7)$$

In this equation, we first calculate the minimum of the start offsets and add the absolute value of the difference between their ranks to obtain the final selection index $\vec{z}$. Next, we use the indices in $\vec{z}$ to select the corresponding lanes from the first and second vectors of $\vec{t}$ in the first call, as well as from the third and fourth vector in the second invocation. To obtain the final interleaved score, we select either the permutation result from the first selection or the second selection, depending on whether the corresponding index in $\vec{z}$ is strictly less than 128 or not. An example selection process is illustrated in Fig. 3.11.

3.3.5 Maximising SIMD throughput

In order to maximise the data parallel throughput on a SMP, we have recently introduced a *saturated execution* pattern. This pattern allows us to always use all addressable SIMD lanes of the underlying SIMD registers, regardless of the sequence sizes and scoring schemes used.

By default, this is not possible due to the sequence lengths and the explicit values of the score functions, which limit the smallest viable operand type that can be used without causing arithmetic overflow during the computation of the DP matrix. While a 2-byte operand type is usually sufficient for most applications working with short-read sequences, other input data may require larger operand types to compute the score values accurately.

To address this issue, some applications implement a saturation mode [Daily, 2016; Farrar, 2007; Rognes, 2011]. This mode follows an iterative procedure to optimise the SIMD throughput. For each computed alignment instance, the algorithm checks if there was an arithmetic overflow. If no overflow is detected, the resulting alignment is considered correct

and can be reported. However, if an overflow occurs, the alignment instance is rescheduled for a second execution cycle using the next larger operand type.

The process begins by computing all alignments with 1-byte packed SIMD vectors, enabling the full utilisation of all addressable lanes of the SIMD registers. Then, the procedure is repeated for all rescheduled alignment instances using 2-byte packed SIMD vectors, followed by 4-byte vectors, and so on, until all alignment instances have been computed. This approach ensures that the SIMD throughput is optimised for each alignment instance by using the smallest possible operand type without causing arithmetic overflows.

Tiled DP matrix

In contrast to the approach described above, we propose a stable saturation mode that computes vectorised alignments solely using 1-byte packed SIMD vectors, regardless of the lengths of the sequences. To achieve this, we divide the interleaved DP matrix into smaller non-overlapping $b \times b$ submatrices, which we refer to as *b-tile*.

For simplicity, we assume, without loss of generalisation, that the dimensions N and M are both multiples of b such that $N_b = \frac{N}{b}$ and $M_b = \frac{M}{b}$. Having this, we define a *tiled DP matrix*, denoted by $\vec{\mathbf{S}}_b$, as a matrix consisting of N_b columns and M_b rows, where each element is itself a $b \times b$ matrix:

$$\vec{\mathbf{S}}_b[\imath, \jmath] = \begin{bmatrix} \vec{\mathbf{S}}[b\imath, b\jmath] & \vec{\mathbf{S}}[b\imath+1, b\jmath] & \cdots & \vec{\mathbf{S}}[b\imath+b-1, b\jmath] \\ \vec{\mathbf{S}}[b\imath, b\jmath+1] & \vec{\mathbf{S}}[b\imath+1, b\jmath+1] & \cdots & \vec{\mathbf{S}}[b\imath+b-1, b\jmath+1] \\ \vdots & \vdots & \ddots & \vdots \\ \vec{\mathbf{S}}[b\imath, b\jmath+b-1] & \vec{\mathbf{S}}[b\imath+1, b\jmath+b-1] & \cdots & \vec{\mathbf{S}}[b\imath+b-1, b\jmath+b-1] \end{bmatrix}$$

with $\imath \in [0..N_b)$ and $\jmath \in [0..M_b)$.

We employed a nested iteration scheme to compute the tiled interleaved DP matrix $\vec{\mathbf{S}}_b$. The outer loop iterates over the tiles in column-major order, while the inner loop computes the regular vectorised DP algorithm as described in the previous sections. Additionally, we use two additional buffers: $\vec{C}$ caching the last score values of each column (last row of a DP tile), and $\vec{R}$ caching the score values of each row (last column of a DP tile). Each buffer is a collection of smaller arrays over $b+1$ (p,w)-vectors, such that $|\vec{C}| = N_b$ and $|\vec{R}| = M_b$. This is illustrated in Fig. 3.12. In the beginning, these buffers are initialised as usual using the respective gap score function.

Let $\vec{\mathbf{B}} = \vec{\mathbf{S}}_b[\imath, \jmath]$ be the tile that is currently being computed. Before this tile can be processed, we subtract the score value of the first entry from each entry of $\vec{C}_\imath$ and $\vec{R}_\jmath$, i.e. $\vec{C}_\imath[k] = \vec{C}_\imath[k] - \vec{C}_\imath[0]$ and $\vec{R}_\jmath[k] = \vec{R}_\jmath[k] - \vec{R}_\jmath[0]$, where $k \in [0..b]$. This step recalibrates the initialisation buffers to the starting condition where the first cell of the initialisation column and row are set to 0 (refer to Section 2.3.4 to recall the initialisation rules). Additionally, we cache the values $\vec{c} = \vec{C}_\imath[b]$ and $\vec{r} = \vec{R}_\jmath[b]$ after recalibrating the scores and before computing $\vec{\mathbf{B}}$.

After finishing the inner loop and before progressing to the next tile $(\imath, \jmath+1)$, we copy the scores of the last row $\vec{\mathbf{B}}_{*,b-1}$ to $\vec{C}_\imath$ and the last column $\vec{\mathbf{B}}_{b-1,*}$ to $\vec{R}_\jmath$ starting at index 1

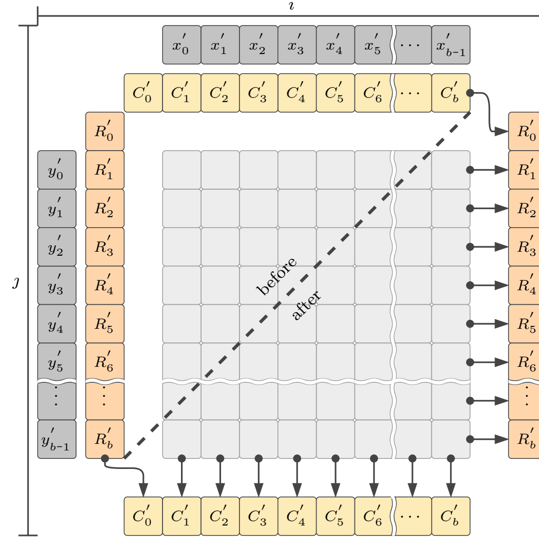


Figure 3.12: Detailed view of a b-tile at $(\imath, \jmath)$. Here $x' = x[\imath b..(\imath+1)b)$, $y' = y[\jmath b..(\jmath+1)b)$, $C' = C_\imath$, and $R' = R_\jmath$. Before the cells of the tile are computed the last values of $C'[b]$ and $R'[b]$ are cached and assigned to the first value of $R'[0]$ and $C'[0]$ respectively after computing the DP tile. The remaining values are copied into the respective cells of C' and R' (black arrows).

respectively. We also set $\vec{C}_\imath[0] = \vec{r}$ and $\vec{R}_\jmath[0] = \vec{c}$, such that the first entry of $\vec{C}$ corresponds to the cached entry of the first row and the first entry of $\vec{R}$ corresponds to the cached entry of the first column (see Fig. 3.12).

Since a tile stores only relative score values instead of absolute ones, we track the cumulative sum of the subtracted offsets for each tile column and row of $\vec{\mathbf{S}}_b$. This is done to recalibrate the column and row initialisation buffers. After all tiles of $\vec{\mathbf{S}}_b$ have been computed, we can use the tracked offsets to recompute the absolute values of the relative scores stored in the respective initialisation buffers. This is done by subtracting the tracked offset sum from the values. In Fig. 3.13, we demonstrate the saturated computation with 4 tiles using the scalar global alignment example from Fig. 2.7a.

Saturated DP tiles

We call a b-tile whose maximal score range is guaranteed to be $[-128..127]$ a *saturated b-tile*. The primary challenge now is to find a suitable value for b such that the b-tiles can be computed with a 1-byte operand type (i.e. `int8_t`) without inducing a score overflow. This can be determined by resolving the following two equations to b.

$$127 \geq b\sigma_{\max} - (g_{\text{opn}} + bg_{\text{ext}}) \tag{3.8a}$$

$$-128 \leq \max \begin{cases} 2(g_{\text{opn}} + bg_{\text{ext}}) \\ b\sigma_{\min} \end{cases} \tag{3.8b}$$

where $\sigma_{\max}$ is the maximal and $\sigma_{\min}$ is the lowest score value of the underlying substitution score model σ, e.g. 11 and −4 in case of the BLOSUM62 matrix or $m_=$ and $m_{\neq}$ in case

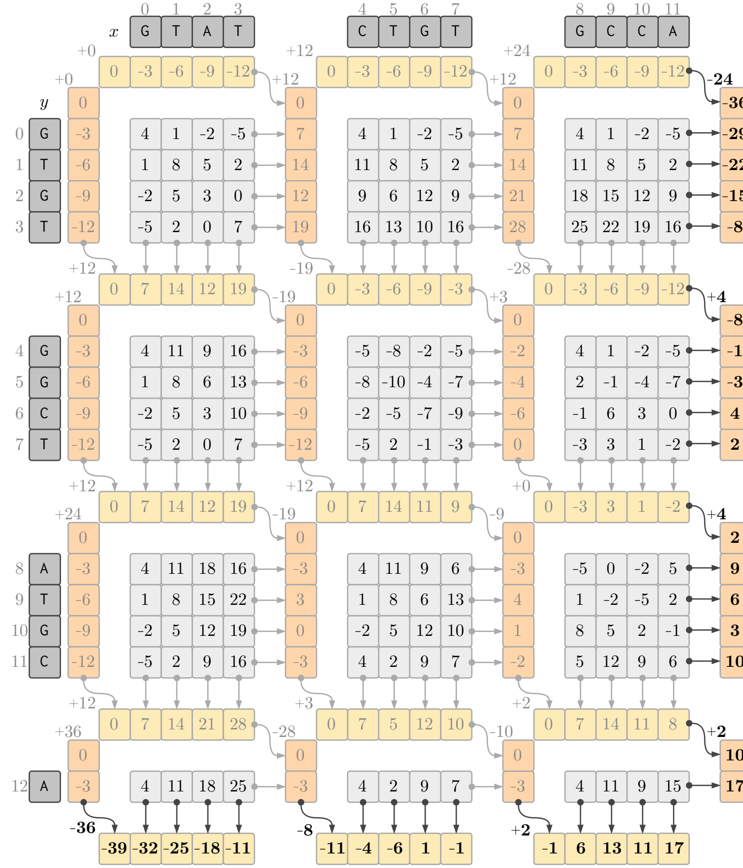


Figure 3.13: Computing the DP matrix of Fig. 2.7a with relative scores using 4-tiles. The yellow and orange entries represent the scores of C and R respectively. Before computing a tile, the scores of the corresponding slice in C and R are reset by subtracting the score stored in their first entry (light grey values at the top left corner of each tile buffer). Absolute alignment scores are recovered by adding the cumulative sum of the subtracted offsets within the same column or row to the relative scores (black scores in last column/row buffer).

of unitary substitution scores. Equation (3.8a) constrains the maximum value of b such that all score values in a tile are less than or equal to 127. Similarly, the second equation constrains the maximum value of b such that all score values in a tile are greater than or equal to -128. Finally, we choose b as the minimum of the two results. Please note that for a concise description, we assume a single scalar DP matrix in the following, but the presented ideas can be transferred one-to-one to our interleaved vectorisation approach.

Theorem 3.3.1. Given a b-tile $\mathbf{B} = \mathbf{S}_b[\imath, \jmath]$, then we guarantee by solving the linear equations Eq. (3.8a) and Eq. (3.8b) and choosing b as the minimum of both that $-128 \leq \mathbf{B}_{i,j} \leq 127$ for all $i \in [0..b)$ and $j \in [0..b)$ in the case of global alignments.

Proof. To see that this is true, consider the sketch depicted in Fig. 3.14. The upper bound (Eq. (3.8a)) is obtained from the observation that after recalibrating the buffers we have $C_\imath[0] = 0$ and $R_\jmath[0] = 0$. By the DP recursion we also have that $c = C_\imath[b] \geq g_{\text{opn}} + b g_{\text{ext}}$ (red arrow) and analogously $r = R_\jmath[b] \geq g_{\text{opn}} + b g_{\text{ext}}$ (green arrow). The largest score

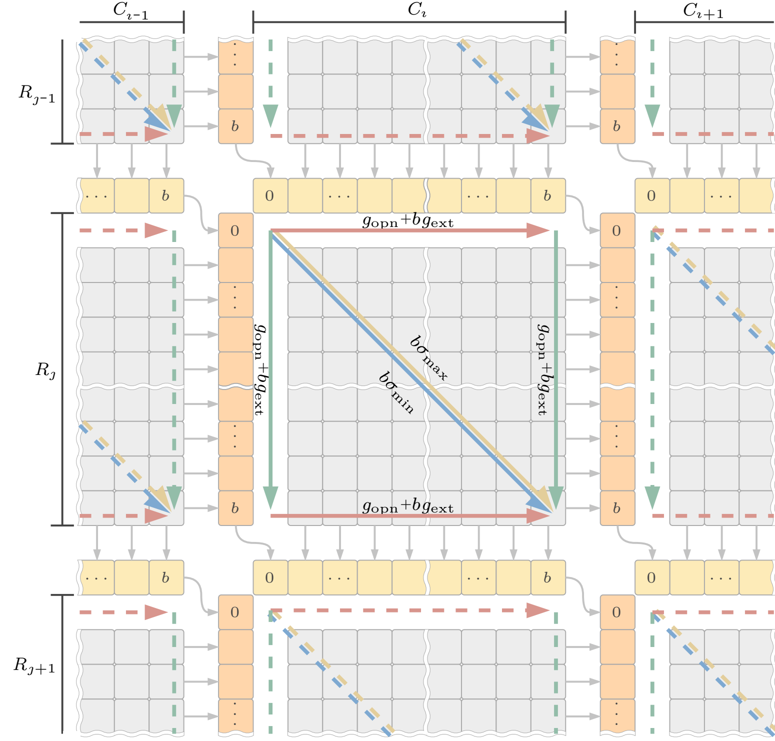


Figure 3.14: Maximal and minimal offsets of a b-tile. The maximum score value within a b-tile is obtained from adding b times the maximal match scores (yellow arrow along major diagonal). The minimum score value is either obtained from deleting the entire segment of x (red arrows) and inserting the entire segment of y (green arrows) or replacing all symbols (blue arrow along the major diagonal). Since we choose the maximum when computing a cell, the larger value of both cases determines the minimum.

value computed within $\mathbf{B}$ is by adding b times $\sigma_{\max}$. Thus, when recalibrating $C_\imath$ before computing the next b-tile $(\imath, \jmath+1)$, we have $C_\imath[b] = C_\imath[b] - C_\imath[0]$ such that

$$C_\imath[b] \leq b\sigma_{\max} - (g_{\text{opn}} + bg_{\text{ext}}),$$

which yields the largest positive score value that must be less than or equal to 127 to avoid an arithmetic overflow. To see that Eq. (3.8b) is sufficient to determine the lower score bound, we need to show that the largest negative score obtained from the recalibration step is greater than the maximum of $2(g_{\text{opn}} + bg_{\text{ext}})$ and $b\sigma_{\min}$. We proof this by contradiction.

Again, the largest negative score is computed during the recalibration step when we can have in $C_\imath[0]$ a score that is as large as possible and in $C_\imath[b]$ a score that is as low as possible. Thus, assume that $C_\imath[b]$ has the score $b\sigma_{\max}$ after computing the previous b-tile at $(\imath, \jmath-1)$. At the same time we have by the DP recursion that $C_\imath[0] \geq \max(g_{\text{opn}} + bg_{\text{ext}}, b\sigma_{\min})$. Let us assume that $g_{\text{opn}} + bg_{\text{ext}}$ is the maximum. After recalibrating C we have $c = C_\imath[b] = b\sigma_{\max} - (g_{\text{opn}} + bg_{\text{ext}})$. Following the computation of $\mathbf{B}$ the lowest computed score can only be derived from following a red and green arrow or the blue arrow. Correspondingly, the updated scores in $R_\jmath$ are

$$R_\jmath[0] = c = b\sigma_{\max} - (g_{\text{opn}} + bg_{\text{ext}})$$

and

$$R_J[b] = \max \begin{cases} 2(g_{\text{opn}} + b g_{\text{ext}}) \\ b\sigma_{\text{min}}. \end{cases}$$

From this follows that

$$\begin{aligned} R_J[b] &= R_J[b] - (b\sigma_{\text{max}} - (g_{\text{opn}} + b g_{\text{ext}})) \\ &= R_J[b] + g_{\text{opn}} + b g_{\text{ext}} - b\sigma_{\text{max}} \\ &\geq R_J[0] + g_{\text{opn}} + b g_{\text{ext}} \\ &= b\sigma_{\text{max}}. \end{aligned}$$

This is a contradiction because by definition of scoring schemes the left hand side of the relational operator $\leq$ is always strictly less than 0 and the right hand side is always greater than or equal to 0. It follows, that due to the recursion formula the lowest score can only be computed within a b-tile and not during the recalibration. $\square$

Saturated interleaved alignment

To apply the saturated execution pattern to our interleaved scoring matrix $\vec{\mathbf{S}}$ computing an optimal global alignment, we need to handle the cumulative offset sums obtained by subtracting the cached $\vec{c}$ and $\vec{r}$ values in each row and column of $\vec{\mathbf{S}}_b$.

Before adding the offsets, we upcast the $(1, w)$-vector to four $(4, w)$-vector, which can be efficiently done using SIMD instructions. As a result, we can use regular 32-bit integers to keep track of the subtracted offsets, allowing us to compute any alignment instance without risking score overflows.

For the local alignment problem, there is an additional issue because absolute scores close to 0 cannot be recalibrated, as the scores are bounded below by 0. In this case, before computing a tile $\vec{\mathbf{B}}$, we check whether the score offsets $\vec{c}$ and $\vec{r}$ being subtracted from the buffers are larger than a precomputed threshold in their absolute representation.

More specifically, before recalibrating the buffers of $\vec{\mathbf{B}}$, we compute the absolute value representation of $\vec{c}$ and $\vec{r}$ using the previously described upcast to larger operand types. If none of the absolute scalar operands is less than $\max(2(g_{\text{opn}} + b g_{\text{ext}}), b\sigma_{\text{min}})$, then we treat $\vec{\mathbf{B}}$ as if it were part of a global alignment, because we are guaranteed to not cut score values less than 0.

In the other case, where the absolute scalar operands are less than the threshold, we do not recalibrate the offsets and reset the cumulative sum of the tracked column and row offsets to 0. Now the relative scores correspond one-to-one to their absolute values again, and we use the regular local alignment procedure to compute this tile.

Maximising saturated tile dimensions

We can further increase the dimension of b by substituting the natural zero value, 0, with an offset that shifts the center of the computed score range. This offset can be shifted towards -128, if the negative score range is not exhausted. Alternatively, it can be shifted towards 127, if the positive score range is not exhausted. To compute the artificial zero offset we adapt the equations shown in Fig. 3.14 by adding the variable z representing the unknown offset at $C_i[0]$ and $R_j[0]$

$$127 \geq z + b\sigma_{\max} - (g_{\text{opn}} + bg_{\text{ext}}) \tag{3.9a}$$

$$-128 \leq z + \max \begin{cases} 2(g_{\text{opn}} + bg_{\text{ext}}) \\ b\sigma_{\min} \end{cases} \tag{3.9b}$$

and solve the linear equation system to obtain the ordered pair (z, b). Doing so for a scoring scheme with affine gap scores $g_{\text{opn}} = -10$ and $g_{\text{ext}} = -1$ and with $\sigma_{\max} = 5$ and $\sigma_{\min} = -4$ would increase b from 19 to 28 by using $z = -51$ as the new artificial value for zero. In the case of the BLOSUM62 matrix for example ($\sigma_{\max} = 11$; $\sigma_{\min} = -4$), this would increase b from 9 to 16 using $z = -75$ as the new zero value. Due to an increased block size, the absolute number of post- and preprocessing instructions needed to keep track of the cumulated offsets is reduced and the overall performance gets improved even further.

3.4 Utilising thread-level parallelism

In this section, we first focus on the central design of dynamically scheduling and executing a bulk of independent AIs in parallel on multiple threads. Afterwards, we refine the introduced scheduling mechanism by extending it with a version that also allows computing a single AI in parallel using the intra-sequence execution pattern. Lastly, we discuss recent extensions to our approach aimed at fully exploiting the available hardware resources.

3.4.1 Dynamic scheduling and execution

The core of our universal parallelisation approach is a two-stage scheduling system consisting of an *alignment scheduler* and an *alignment executor*.

In the first stage, the alignment scheduler receives and manages asynchronously submitted AIs. This can be either a single AI or a bulk of AIs. Underneath, the alignment scheduler uses a *multiple producer, multiple consumer* (MPMC) *concurrent queue* in combination with a *thread pool* to orchestrate the concurrent execution of the submitted AIs. A thread of the alignment scheduler, which is the consumer of an AI, is responsible for managing the execution of its associated AI but does not perform the actual work itself. Instead, it packages the given AI into an AT and asynchronously submits it to the downstream alignment executor.

Meanwhile, the managing thread from the alignment executor, which owns the respective AI, waits until the task is completed within the alignment executor. Similar to the alignment scheduler, the alignment executor utilises its own thread pool together with an MPMC concurrent queue to asynchronously handle incoming ATs. When a thread of the alignment executor fetches a new AT from the accompanying queue, it immediately starts executing

Figure 3.15: Sequence diagram illustrating the concurrent interactions between a user, the alignment scheduler, and the alignment executor. The scheduler and executor set up a thread pool with two concurrently running threads each ($T0$, $T1$; and $T2$, $T3$) and a concurrent queue. The user asynchronously submits 2 single AIs and one bulk AI covering 4 individual AIs. After the scheduler is initialised, $T0$ and $T1$ start spinning, waiting for an AI to become available. $T1$ pops the first AI from the queue, while $T0$ takes the bulk AI. Both threads create an Alignment Task (AT) for the received AI and submit it to the concurrent queue of the alignment executor. After submitting the ATs, the threads are suspended (orange lifeline). On the side of the alignment executor, $T2$ processes the AT submitted by $T1$, and $T3$ processes the AT submitted by $T0$. Once they complete their tasks, the corresponding threads of the alignment scheduler ($T1$ first, and $T0$ second) are notified. Once these threads are awakened, they perform postprocessing tasks such as tracing the alignment and notifying the user when the result is ready. Afterwards, they can process the next AIs or wait until one becomes available.

it. Once the task is completed, the corresponding thread tries to fetch the next AT from the queue or waits until another AT becomes available. This general two-stage scheduling system is illustrated in Fig. 3.15.

Thread suspension

Threads of the alignment scheduler utilise a passive blocking strategy by setting up a *condition variable* during the initialisation of the respective AT. Specifically, a thread of the alignment scheduler is likely suspended by the operating system after successfully submitting the AT. In this case, they do not waste vital CPU resources while waiting for the alignment task to be computed.

Once the corresponding AT finishes in one of the threads of the alignment executor, it notifies the associated condition variable, signalling the completion of its work. Upon waking up, the alignment scheduler thread produces the final alignment result and sends it back to the user through a given callback function, allowing the user to handle the results asynchronously if desired.

Subsequently, a thread of the scheduler checks if more AIs are available in the concurrent queue and either extracts the next one in the queue (Thread 1 in Fig. 3.15) or waits for one to become available.

Once all AIs are submitted, the user can explicitly close the alignment scheduler to indicate that no more AIs can be submitted. The closing signal is also propagated to the linked alignment executor. If both the scheduler and executor are closed and their queues are empty, the concurrently running threads will terminate and safely clean up all thread-local resources allocated during the execution.

Chunked scheduling

The operations to yield and wake a thread are handled entirely by the operating system and may introduce some extra overhead that can negatively impact overall performance. This can become problematic, especially when dealing with alignment requests consisting of many small-sized AIs, where the amount of work per AT is too small and the multithreading overhead limits the performance gains.

To address this, we implemented a chunking mechanism where threads of the alignment scheduler extract not just one, but a chunk of submitted AIs and bundle them into a single bulk AT. This allows us to control the amount of work per execution within the alignment executor, reducing the relative overhead compared to the total execution time. For example, in Fig. 3.15, the user submits a bulk of four AIs in a second call, which are pushed to the underlying queue. Then, Thread 0 dequeues all four instances at once and wraps them into an AT responsible for computing all four alignments.

Notably, we made the maximum size of the chunks, as well as the number of threads in each stage, customisable in order to accommodate different alignment request scenarios and provide an optimal load balancing strategy for each case. Additionally, the threads of the

alignment scheduler and executor can be programmatically pinned to specific cores, giving the user full control over the scheduling and execution system.

3.4.2 Intra-sequence execution

In its basic form, the presented two-stage scheduling mechanism is designed for the inter-sequence execution pattern. To enable intra-sequence execution using this scheduling system, we refined the original mechanism to create a tiled DP matrix (see Section 3.3.5) for a received AI, similar to the approach used for computing an interleaved DP matrix with saturated tiles. However, in this case, we are not limited by the size of b and can use much larger values.

Task graph representation

To execute the obtained tiles in parallel, we wrap each DP matrix tile into a separate AT and generate a *task graph* $G_b = (V, E)$ where each AT is represented by a node. Correspondingly, we have a tiled DP matrix $\mathbf{S}_b$, consisting of $n_b = \frac{n}{b}$ columns and $m_b = \frac{m}{b}$ rows, for a given AI (x, y), with $x = \Sigma^n$ and $y = \Sigma^m$. Note we assume for simplicity that without loss of generalisation n and m are a multiple of b. As a result the node set V of the task graph G_b contains $n_b \cdot m_b$ many nodes.

Two nodes in the task graph G_b are connected with a directed edge if the corresponding tiles in $\mathbf{S}_b$ are adjacent. More specifically, let u and v be two nodes of V such that $\mathbf{S}_b[\imath, \jmath]$ is the b-tile represented by u and $\mathbf{S}_b[\imath', \jmath']$ is the b-tile represented by v. Then there exists a directed edge $e = (u, v) \in E$ if and only if $\imath' = \imath + 1$ and $\jmath' = \jmath$, or $\imath' = \imath$ and $\jmath' = \jmath + 1$. According to this setup, the root node of G_b represents the task to compute the b-tile $\mathbf{S}_b$ [0, 0] and is characterised by having no incoming edges. Furthermore, all tiles on the first column and row are represented by nodes with exactly one incoming edge, while all other nodes have two incoming edges. Figure 3.16 depicts a task graph build over a tiled DP matrix.

Task graph execution

The edges in the task graph G_b represent natural synchronisation points for the parallel execution of the associated ATs. That is, the task of a node $v \in V$ can only be executed after all its incoming edges have been visited. We model this by augmenting each node with an *atomic counter* that initially equals its in-degree. After completing the AT, the active thread runs a postprocessing routine that follows the outgoing edges of the currently processed node and decrements the atomic counter of the successor nodes. If the dependency count of a successor node reaches 0, the same thread submits the AT associated with that node to the concurrent queue of its linked alignment executor.

To initiate the execution, the alignment scheduler only submits the root node of the generated task graph to the alignment executor. From that point on, the correct execution order of the associated ATs is completely handled by the graph structure within the context of the alignment executor. The resulting execution pattern resembles a dynamic wavefront progressing along the minor diagonals of the tiled DP matrix, rather than strictly following

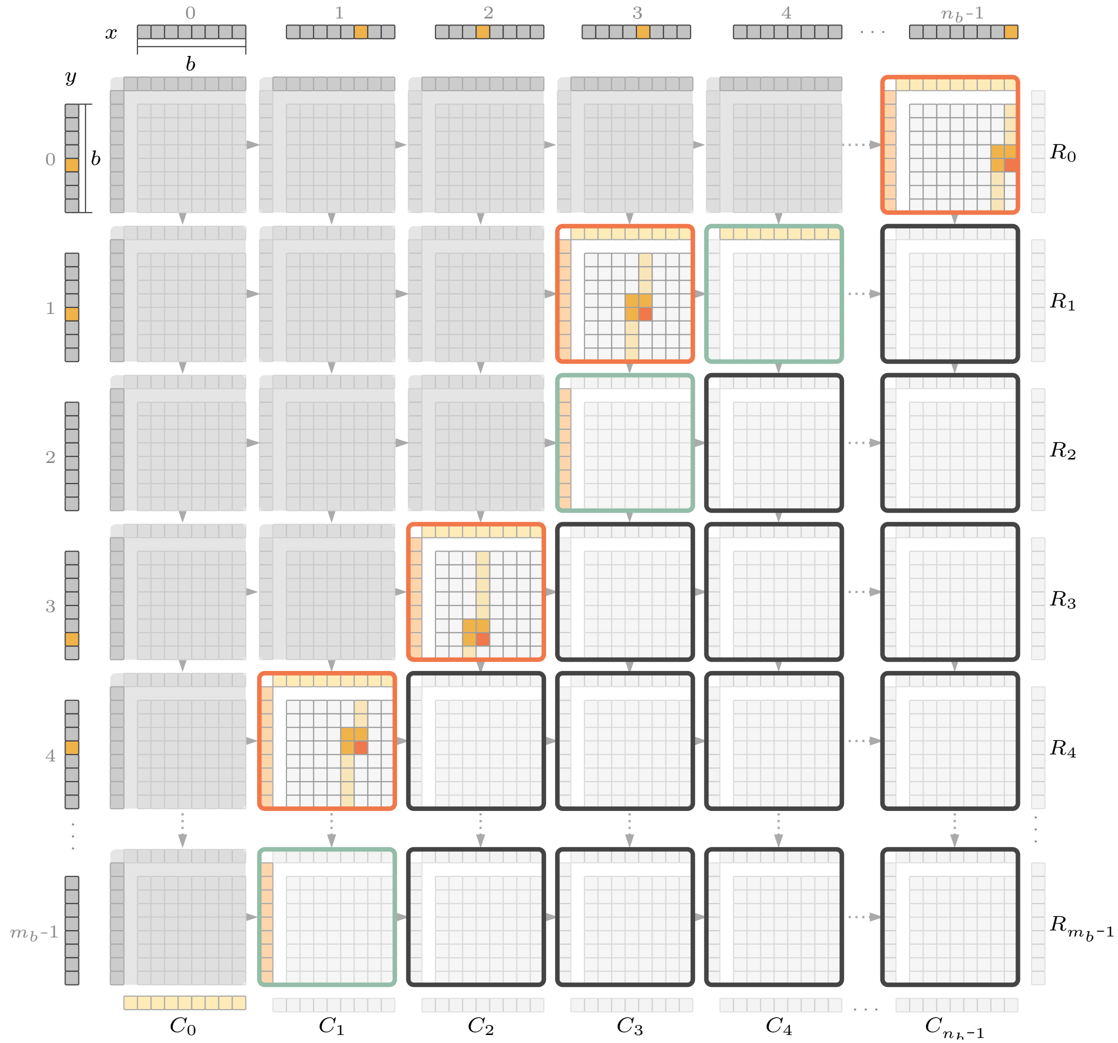


Figure 3.16: Task graph build over a tiled DP matrix. Each column and row of the tiled DP matrix represents a non-overlapping segment of length b of x, respectively y, and the corresponding buffer of size $b + 1$ from C and R. Each tile is wrapped in a task graph node and constitutes a separate AT. Arrows between the nodes represent task graph dependencies. Nodes that have been already computed are covered in grey. Nodes with orange lines are currently computed simultaneously $((1, 4), (2, 3), (3, 1)$, and $(n_b - 1, 0))$. Nodes with green lines $((1, m_b - 1), (3, 2)$, and $(4, 1))$ have been already visited once and nodes with black lines have been not visited so far.

the minor diagonals in a rigid manner. An example of a possible execution flow is shown in Figure 3.16, where four nodes are currently being processed, and each thread is in a different state of computing the associated partial AI.

Synchronisation of alignment tasks

Similar to how we organised the computation of the tiles in the saturated vectorisation mode, we allocate two initialisation buffers, C and R, of size n_b and m_b, respectively. Each buffer consists of an array containing $b+1$ score values that are used to populate the scores for the next tile in the same column or row. These initialisation buffers are shared among all threads of the alignment executor since the order of how the associated ATs are executed and which thread executes them is not deterministic. However, due to the design of the task graph using atomic counters to model the dependencies, we do not require any special read or write protection for updating these buffers. In fact, only one tile can be actively computed by a thread on a particular column $\imath$ and row $\jmath$ using this approach.

Moreover we ensure that each local buffer starts on a new cache line such that we minimise performance bottlenecks due to maintaining cache coherency. Figure 3.12 illustrates how these buffers are used to initialise a tile before computation and how they are updated after the computation of the tile is finished in order to obtain the correct alignment scores.

The AT computing the last tile of $\mathbf{S}_b$, i.e. $\mathbf{S}_b[n_b - 1, m_b - 1]$, stores an additional condition variable. It can use this variable to notify the waiting thread of the alignment scheduler that initialised and triggered the execution of the task graph. Once awakened, the thread issues the postprocessing step of the alignment algorithm.

To facilitate this step, we implemented a thread-local alignment result buffer that is shared among all threads of the alignment executor and the alignment scheduler. This result buffer contains a slot for each thread of the alignment scheduler, and each slot has an array that stores the intermediate results produced by each thread of the alignment executor while executing an AT from that particular scheduling thread.

During the postprocessing, the managing thread can access all intermediate results to generate the final result. This could be, for example, the optimal score tracked by each individual thread of the alignment executor in the case of computing a local alignment, or fragments of the trace matrix. Once the postprocessing is done, the final alignment result is sent to the user.

Bulk execution

In order to fully utilise all available hardware resources of an SMP, we combined the task graph execution approach with our vectorised inter-sequence execution pattern. When there are enough ATs submitted to the concurrent queue of the alignment executor, we collect not just one, but a bulk of ATs from the queue (e.g. Thread 3 in Fig. 3.15). Each thread in the alignment executor tries to fetch $l = \frac{w}{p}$ ATs in order to fully utilise an interleaved DP matrix configured for (p,w)-vectors. If not enough ATs are available the executing thread proceeds with just one AT using the respective scalar DP implementation. This allows us

to fully utilise all lanes of the VPU registers, while also not stalling at the begin or the end of an executed task graph where the number of concurrent tiles may be low.

3.5 Improving cache efficiency and instruction-level parallelism

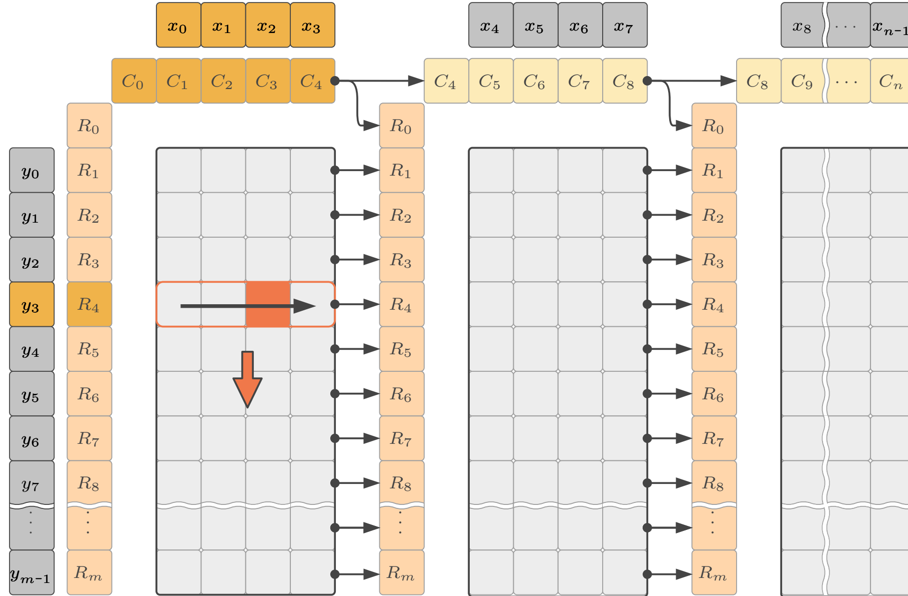


Figure 3.17: Striped execution of a DP matrix. Instead of computing a single cell per column, s consecutive cells in the same row are computed one after another (dark arrow at row 5) before going to the next row in the same stripe (big orange arrow). Here s is 4 and the third cell of the fourth row of the first stripe is currently computed.

While profiling the alignment algorithms, we observed that updating the scores of the current entry in the row buffer R took significantly more time compared to updating the scores of the current entry in the column buffer C. Upon further investigation of the execution profile, we found that these values are directly read from and written to the corresponding memory address in R, resulting in additional overhead due to the latency of memory loads, while the values of the C were cached.

3.5.1 Striped execution pattern

To mitigate this overhead, we implemented a *striped* iteration pattern over the DP matrix by grouping columns of the DP matrix into consecutive, non-overlapping stripes of constant width $s \in \mathbb{N}$. Conceptually, a stripe represents a $s + 1 \times m + 1$ submatrix of the original DP matrix. With this revised execution pattern, we process one stripe at a time using a row-major traversal within each stripe (see Fig. 3.17).

The main advantage of this revised execution pattern is that the loaded elements of the query (i.e. y) sequence and the corresponding element of the row buffer R are cached for as long as algorithm stays in the same row. Since s is a small constant number the accessed entries of the column buffer C and the elements of the reference sequence (i.e. x) are cached as well. For example, in Fig. 3.17, we divide the costs of loading y_3 and R_4 by four. After the

initial values are loaded into the CPU registers, we cache all intermediate results between the columns in temporary variables, which are likely to be stored in the first cache level, thus benefiting from low latencies. Only after computing an entire stripe row, the final R value is written back to the corresponding memory address.

3.5.2 Loop unrolling

To further improve the performance, we unrolled the loop to compute the row of a stripe giving the compiler the opportunity to pack the instructions for computing multiple cells more effectively (see Section 3.1.5). However, instead of hard-coding the loop unrolling we used an automated approach by utilising *fold expressions*[4] which is a C++17 language feature.

Before computing a stripe we prefetch the corresponding elements of x and C into static arrays (i.e. `std::array`) of size s. For each row j we cache the currently accessed values y_{j-1} and R_i. Then we update the s cells of this row by letting the compiler unpack the prefetched arrays. By unpacking the cached array elements and applying the function that computes a single cell, the loop invariants for a single row in the stripe are eliminated. Furthermore, the compiler could optimise the instructions collectively, resulting in improved utilisation of instruction-level parallelism.

It is important to note that this striped execution pattern is independent of whether scalar or interleaved scores are used, as well as the specific function employed to compute a single cell. Consequently, we integrated this feature into the base implementation of the DP matrix, recognising that the width of the stripe is typically smaller than the tile dimension used in the saturated SIMD execution pattern.

3.6 Applications and evaluation

In this section, we will assess the effectiveness of our methodologies in enhancing the performance of DP algorithms for specific alignment requests commonly encountered in various bioinformatics applications. We will evaluate the following scenarios:

- The computation of millions of alignment instances using short homogeneous DNA sequences (SoBA). This scenario is typical in read mapping applications that process short-read sequencing data.
- The computation of thousands of sequence alignments using heterogeneous DNA sequences with varying lengths, ranging from a few hundred bases to a few kilobases (LoBA). This scenario arises when dealing with long-read sequencing data.
- The computation of single pairwise alignments for large-scale DNA sequences with sizes in the megabase range (LoSA).

Additionally, we will compare the recent improvements we made to the vectorised inter-sequence execution pattern with our previous implementation. Finally, we will discuss the overall design of our implementation and provide insights into the new API in the modernised version of the SeqAn software library.

[4] `https://en.cppreference.com/w/cpp/language/fold`; accessed 20.02.2023

3.6.1 Test systems

	HSW	SKX	ICX	KNL
code name	Haswell	Skylake	Icelake	Knights Landing
model	Intel® Xeon® E5-2650 V3	Intel® Xeon® Gold 6148	Intel® Xeon® Platinum 8358	Intel® XeonPhi™ 7250
sockets	2	2	2	1
physical cores	20 (10 per socket)	40 (20 per socket)	64 (32 per socket)	68
logical cores	40	80	128	272
base frequency	2.3 GHz	2.4 GHz	2.6 GHz	1.4 GHz
SIMD ISA	SSE4, AVX2	SSE4, AVX2, AVX512F, AVX512BW	SSE4, AVX2, AVX512F, AVX512BW, AVX512VBMI	SSE41, SSE42, AVX2, AVX512F
SIMD register	16	32	32	32

To evaluate the effectiveness of our generic accelerated alignment module, we conducted tests on four different microprocessor architectures, referred to by their codenames: Haswell (HSW), Skylake (SKX), Icelake (ICX), and Knights Landing (KNL). In the following discussion, we will use the abbreviated notation for each test system.

The key attributes of these systems, relevant to our analysis, are summarised in Section 3.6.1. The first three processors belong to Intel's scalable Xeon® processor family. HSW supports an SIMD ISA up to AVX2, with sixteen 256-bit registers per physical core. SKX supports AVX512 ISA, and it provides the relevant AVX512BW instructions for one and two-byte operand types. With ICX, the additional AVX512VBMI ISA, which includes `vpermi2b` instructions for shuffling one-byte operand types from two 64-packed SIMD registers.

The last system we tested is the XeonPhi® coprocessor that can also operate as a normal CPU hosting the operating system. This system first introduced AVX512 instructions, but only supporting relevant instructions for 4 and 8-byte operand types. Although Intel has discontinued the XeonPhi® processors, we included this system in our evaluation to demonstrate scalability with many-core systems and to showcase the applicability of our methods across different CPU architectures.

All systems were running a linux operating system. We used g++-10.3.0 to compile the binaries for the evaluation. All benchmark applications were compiled in release mode with `-O3 -DNDEBUG`. If possible we fixed all processors to their respective base frequency.

3.6.2 Evaluation of short bulk alignment requests

To evaluate the performance of our implementation given a SoBA request we simulated Illumina single-end reads with a length of 150 bases using Mason in version 2.0.8 [Holtgrewe, 2010]. This test data set included altogether 12 497 500 alignment instances for which we computed the optimal global (GLO), semi-global (SGL), overlap (OVL), and local (LOC) alignment using the regular algorithm and the k-band version on HSW, SKX, and KNL. We used unitary substitution costs with $m_{=} = 5$ and $m_{\neq} = -4$, and affine gap costs with

SIMD-ISA (#lanes)		seqan			seqan + band			parasail		
		GLO	SGL	LOC	GLO	SGL	LOC	GLO	SGL	LOC
HSW (20 threads)										
SSE4 (8)	time [s]	5.90	5.90	7.38	1.39	1.36	1.47	8.93	8.45	8.34
	GCUPS	48.28	48.27	38.60	23.13	23.54	21.76	31.91	33.71	34.17
	speedup	1.51	1.43	1.13	6.44	6.20	5.66	1.00	1.00	1.00
AVX2 (16)	time [s]	2.80	2.76	3.48	0.85	0.85	0.91	9.40	8.87	8.12
	GCUPS	101.90	103.08	81.86	37.85	37.81	35.32	30.33	32.11	35.11
	speedup	3.19	3.06	2.40	10.54	9.96	9.18	0.95	0.95	1.03
SKX (40 threads)										
SSE4 (8)	time [s]	2.68	2.68	3.26	0.62	0.61	0.74	3.88	3.66	4.11
	GCUPS	106.27	106.26	87.43	51.33	52.54	43.27	73.40	77.86	69.41
	speedup	3.31	3.34	2.66	14.22	14.65	11.67	2.17	2.16	1.99
AVX2 (16)	time [s]	1.35	1.35	1.55	0.40	0.43	0.42	4.00	3.88	3.96
	GCUPS	211.32	211.47	183.76	79.46	74.57	76.34	71.21	73.43	71.68
	speedup	6.59	6.64	5.58	22.02	20.80	20.60	2.10	2.17	2.14
AVX512 (32)	time [s]	0.68	0.68	0.80	0.24	0.25	0.26	N/A	N/A	N/A
	GCUPS	420.41	420.42	354.44	135.53	128.07	125.13	N/A	N/A	N/A
	speedup	13.11	13.20	10.77	37.55	35.72	33.76	N/A	N/A	N/A
KNL (68 threads)										
SSE4 (4)	time [s]	10.81	10.81	14.96	1.64	1.64	2.09	11.67	11.64	12.19
	GCUPS	26.36	26.36	19.05	19.52	19.55	15.33	24.41	24.48	23.38
	speedup	0.82	0.83	0.58	5.41	5.45	4.14	0.77	0.77	0.71
AVX2 (8)	time [s]	4.11	4.11	6.35	0.93	0.97	1.07	13.53	13.75	15.39
	GCUPS	69.33	69.35	44.84	34.48	32.91	29.96	21.06	20.73	18.51
	speedup	2.16	2.18	1.36	9.55	9.18	8.08	0.66	0.65	0.56
AVX512 (16)	time [s]	2.35	2.34	2.53	0.75	0.77	0.84	N/A	N/A	N/A
	GCUPS	121.45	121.73	112.55	42.98	41.58	38.04	N/A	N/A	N/A
	speedup	3.79	3.82	3.42	11.91	11.60	10.26	N/A	N/A	N/A

Table 3.4: Performance comparison of computing 12 497 500 pairwise alignments on different processors (HSW, SKX, KNL) with different SIMD-ISAs (SS4, AVX2, AVX512), alignment problems (glo: global; sgl: semi-global; loc: local), and DP implementations (seqan, seqan + band, parasail). The alignment instances consisted of DNA sequences with a fixed size of 150 bases. We used unitary substitution costs with $m_= = 5$ and $m_{\neq} = -4$, and affine gap costs with $g_{\text{ext}} = -1$ and $g_{\text{opn}} = -10$. For the k-banded execution we used a band of size $k = 4$. The numbers behind the ISA architectures represent the number of utilised lanes.

$g_{\text{ext}} = -1$ and $g_{\text{opn}} = -10$ in all experiments. The band was configured with a band width of 16 (i.e. $k = 8$) to represent an error rate of 5%.

If applicable, we compared our implementations with Parasail in version 2.0.2 which was the most recent version available at the time of the evaluation. We tried all possible alternative inter-sequence execution modes that Parasail offered and chose the one with the best results. Note, that Parasail supports SIMD vectorisation only up to AVX2, such that there is no data available for AVX512 and it also does not support vectorised banded alignments. We also tried available applications based on Libssa and Opal, however, both tools only work for protein sequences and only facilitate the data base search problem, such that we had to exclude them from the benchmarks.

The primary results for this setting are presented in Table 3.4. On each system we used the maximal number of physical cores. The first column represents the used SIMD ISA and the number of utilised lanes per SIMD instruction. For these experiments, we used 16-bit integers as the scalar score type. For each combination we list the execution time in seconds, the *Giga Cell Updates Per Second* (GCUPS) which represents the number of DP matrix entries updated per second, and the speedup factor that is based on the execution time of Parasail using the SSE4/8 configuration for each executed alignment problem. Note the execution times for computing the overlap alignment were in all experiments almost identical to the ones of the semi-global alignment. Thus, we left out the results for this alignment option to keep the table more readable. First, we compare our results with the Parasail library, and afterwards evaluate the results with respect to the different architectures.

Intra- vs inter-sequence execution pattern

In general, our generalised inter-sequence execution pattern outperforms Parasail in all but one case, which is the local alignment for SSE4 on the KNL (factor 0.58 vs. 0.71). Furthermore, our scheme scales perfectly with larger register sizes, while Parasail seems to be limited for this kind of data, since execution times only scale with the number of threads but did not further decrease when increasing the data-parallelism with AVX2 ISA. Note AVX512 is currently not supported by Parasail, so that we could not compare to it. In contrast to this the results indicate that our method scales perfectly with more cores and larger data-parallelism. Thus, we could double the peak performance by increasing the data-parallelism with larger register sizes on all systems.

Banded and non-banded alignment

The banded alignment was the fastest in all experiments, which is expected due to the strong reduction of the entire workload. This also means, that the total overhead induced by the parallel execution (vectorisation and multi-threading) has more effects on the total execution time such that we can observe lower speedup factors compared to the regular computation. The fastest algorithm was the banded version of SeqAn using AVX512/32 on SKX finishing the computation in 0.24s (AVX2: 0.4s; SSE4: 0.61s). The timings for the banded local alignment were similar: AVX512: 0.26s; AVX2: 0.42s; SSE4: 0.74s. The banded version was roughly 3 times faster than the non-banded case for the global alignment

(AVX512: 0.68s; AVX2: 1.35s; SSE4: 2.68s) as well as for the local alignment (AVX512: 0.80s; AVX2: 1.55s; SSE4: 3.26s). Compared to Parasail this is 5 to 17 times faster for the global alignment (AVX2: 4.00s; SSE4: 3.88s) and the local alignment (AVX2: 3.96s; SSE4: 4.11s). In total SeqAn achieves a peak performance of roughly 420 GCUPS, while Parasails peak performance was 77.86 GCUPS. In other words, we can run about 50 million alignments per second with the affine gap model on the respective SKX system.

Many-core architectures

The best performance on the KNL using AVX512 with 32 bit packed integers is slightly better than the best performance measured on the HSW using AVX2 with 16 bit packed integers. In both cases 16 alignments are computed in parallel in one vector. Moreover, both SSE4 and AVX2 do not perform well on the KNL. The AVX2 implementation is by a factor from 1.75 to 2.5 slower than the AVX512 implementation for the non-banded implementation, although one would expect that they would roughly yield the same result. But since there is no native support for 8 bit and 16 bit instructions on the KNL, these instructions might be executed with much higher latency than natively supported instructions. Compared to the SKX, the KNL is slower by a factor from 3 to 3.5. However, on SKX twice as many alignments can be computed in the vector unit due to the AVX512BW ISA and the clock rate is much higher.

3.6.3 Evaluation of long single alignment requests

In this section we evaluate the scalability of the dynamic task-graph implementation for our multi-threaded intra-sequence parallelisation without vectorisation. We aligned Enterobacteria phage SPC35 (Id: NC_015269.1; length: 118 351 bp) with Salmonella phage Shivani (Id: NC_028754.1; length: 120098 bp) and Chlamydia trachomatis D/UW-3/CX chromosome (Id: NC_000117.1; length: 1042519 bp) with Thermotoga maritima MSB8 chromosome (Id: NC_000853.1, length: 1 860 725 bp) on the KNL.

Multi-threading performance

The results for different thread counts are shown in Fig. 3.18. For these experiments we iteratively increased the number of threads until all logical cores provided by the KNL system were in use. The dashed blue line represents the optimal scaling curve based on the sequential scalar execution of the alignment over the number of threads. The differently coloured lines represent measurements with different partitioning levels.

For the smaller alignment the task-graph implementation scales perfectly until 16 threads, independent for all DP matrix partitions (see Fig. 3.18a). After this, coarse-grained partitions started to plateau, while the most fine-grained partition (blue line; tile size 500) scaled nearly perfect until reaching the maximal number of available physical cores, i.e. 68. When aligning longer sequences (see Fig. 3.18b), this effect gets mitigated and the multi-threaded execution scales independent of the tile size. With longer sequences a larger tile size will still produce enough work for the alignment executor, as there are enough tiles along the minor diagonals to utilise all cores effectively. Using more than 68 threads, however, had a

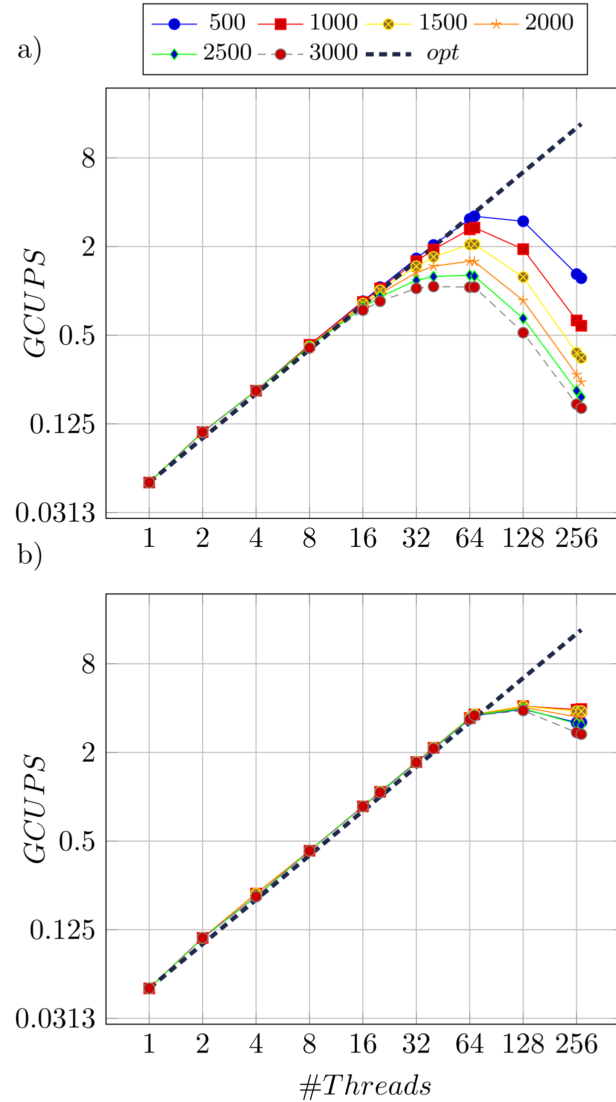


Figure 3.18: Evaluation of the scalability of the non-vectorised wavefront model with different sequence lengths on the KNL. In a) is shown the results for the small sequences of ∼ 120 Kbp length and in b) for the longer sequences larger than 1 Mbp.

negative impact on the performance in both experiments. This can be explained by the fact, that in this experimental setting the 68 physical cores were already fully utilised and thus adding more threads yielded no effective increase of instruction-level parallelism. The observed performance drop was stronger for the shorter sequences and for larger tile sizes, indicating that not enough concurrency could be utilised in this setting.

Combining multi-threading with SIMD parallelisation

In the second experiment we aligned long genomic sequences using the local alignment algorithm by combining the task-graph approach with our interleaved vectorisation approach. The sequence data used for this experiment are the same as used by Liu et al. to evaluate their SWAPHI aligner. The concrete sequences are described in Table 3.5.

ID	Accession No.	Length	Genome Description
DS4.4M	NC_000962.3	4 411 532	Mycobacterium tuberculosis H37Rv
DS4.6M	NC_000913.3	4 641 652	Escherichia coli K12 MG1655
DS23M	NT_033779.4	23 011 544	Drosophila melanogaster chr. 2L
DS33M	BA_000046.3	32 799 110	Pan troglodytes DNA chr. 22
DS42M	NC_019481.1	42 034 648	Ovis aries breed Texel chr. 24
DS50M	NC_019478.1	50 073 674	Ovis aries breed Texel chr. 21

Table 3.5: Large-scale sequences of different sizes used to evaluate the vectorised task-graph alignment.

We computed the local alignments of the alignment instances DS4.4M and DS4.6M, DS23M and DS33M, DS23M and DS42M, DS23M and DS50M, DS33M and DS42M, DS33M and DS50M, and DS42M and DS50M. We conducted the experiments on SKX and KNL using the maximum number of available physical cores for each system. We repeated the experiments with different tile sizes. Moreover, for the experiments on SKX we used a preliminary version of our saturated execution mode, were we used 16-bit integer operand types to compute the relative scores of interleaved tiles and populating the obtained results of the last row and column to the absolute scores before the tiles were submitted to the queue again. Table 3.6 summarises the results for the tile sizes that achieved the best results on the respective processor.

We can observe that for smaller sequences a smaller tile size yields better results. Although D4.4M and D4.6M are over 4 Mbp long, in order to fully utilise the VPUs the number of concurrently executable tiles must scale with the number of SIMD lanes which is 32 on the SKX and 16 on the KNL. Thus, the tile size needed to be reduced in order to produce enough work, so that all threads can execute bulks of submatrices using the interleaved vectorisation. At the same time, setting the block size too small will increase the runtime as the overhead for initialising the interleaved DP matrix, e.g., gathering the respective buffer values from the selected tiles or transforming the sequences into vectors, becomes too large in proportion to the execution time. On the SKX we observed a block size of 3000 to perform best for long sequences giving a peak performance of $\sim$ 258 GCUPS which is 1400 times faster than the sequential execution of the scalar algorithm. On the KNL the peak performance if $\sim$ 80 GCUPs was obtained for tile sizes in the range from 1100 to 1900, which improved the baseline execution time on the KNL by a factor of 2500.

	SKX (t = 40; AVX512/32)			KNL (t = 68; AVX512/16)		
	time [s]	GCUPS	speedup	time [s]	GCUPS	speedup
DS4.4M & DS4.6M	137.61	148.81	816.87	327.57	62.51	1952.91
DS23M & DS33M	3155.64	239.18	1312.96	9487.54	79.55	2485.27
DS23M & DS42M	3831.22	252.47	1385.95	12 009.70	80.54	2516.17
DS23M & DS50M	4601.75	250.40	1374.56	14 605.50	78.89	2464.66
DS33M & DS42M	5327.20	258.80	1420.70	17 477.40	78.88	2464.41
DS33M & DS50M	6384.45	257.25	1412.15	20 374.70	80.61	2518.25
DS42M & DS50M	8147.52	258.34	1418.16	26 426.00	79.65	2488.31

Table 3.6: Performance evaluation of the vectorised task-graph alignment for single large-scale sequence alignments on SKX and KNL using sixteen 16-bit and 32-bit lanes respectively. The shown speedup factor is based on the GCUPS interpolated for the scalar DP algorithm on the respective platform.

3.6.4 Evaluation of long bulk alignment requests

In the this section we evaluate our generalisation to schedule multiple alignment instances concurrently, using an experimental setting in which we align long-read sequences. We used a simulated data set generated with PBSim version 1.0.3 [Ono et al., 2013]. The configuration for which we obtained the data set is depicted in Table 3.7. The simulated data contained 66 860 sequences with the smallest sequence having a length of 2341 bases and the longest 52 668 bases and in average a length of 20 011 bases.

Parameter	Value
reference	GRCH38 chr10
mode	CLR
qc model	default
depth	10
length-mean	20000
length-sd	5000
length-min	100
length-max	60000

Table 3.7: Configuration of PBSim.

Figure 3.19 compares the experimental results running on HSW, SKX, and KNL. In general, execution times on HSW are slightly slower than on SKX using the same ISA and the same number of threads, which results from the slightly slower clock frequency. AVX512 greatly improves the performance, albeit the scaling is with a factor 1.7 not as optimal as observed in Table 3.4. Furthermore, the KNL despite, it's many cores, can only slightly improve the peak performance with respect to HSW. This is due to the fact, that we could only utilise 68 cores of the KNL effectively, while the clock frequency was only 1.4 GHz. Moreover, on the KNL we could not benefit from computing the interleaved submatrices with 16-bit operand types using 512-bit SIMD instructions.

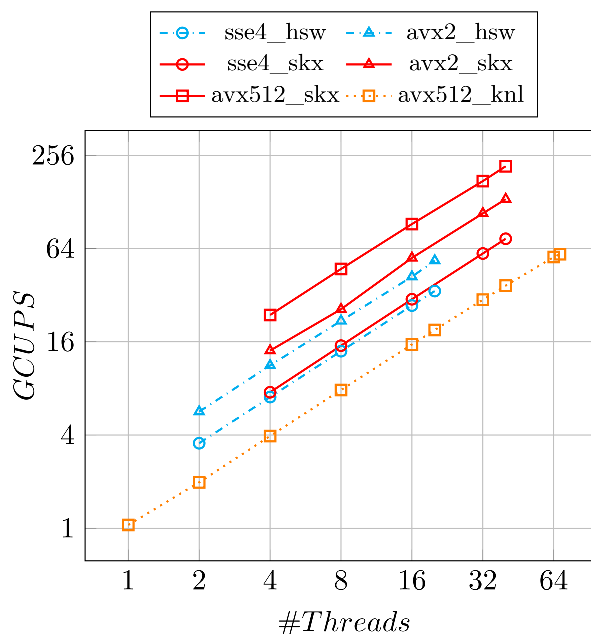


Figure 3.19: Performance and scalability comparison of aligning the PacBio-Sim data on HSW, SKX and KNL.

The measured peak performance was 217 GCUPS on the SKX, 59 GCUPS on the KNL and 53 on the HSW. This is in general lower than the peak performance measured for the large-scale sequence alignments, which can be explained by the more homogenous data of the long sequences. In this case most of the alignment tiles have the same length such that additional bookkeeping for different lengths of the sequences in the vectorised kernel could be omitted. The same observation can be made for the AVX512 results. Since twice as many alignments can be computed in one vector the more likely it is that the tiles executed simultaneously differ in their lengths resulting in an additional overhead for the bookkeeping. Hence, the less heterogeneous the sequence lengths in the data sets are the more efficient is the vectorisation.

We also compared our implementation against Parasail and the *ksw2-test* tool from the *ksw2*-library[5], which is used in minimap2 [Li, 2018] for the alignment computation. Unfortunately, the test tool is only single threaded, but we executed it on the HSW to compare it against our parallelization strategy without multi-threading. In our benchmark ksw2 achieved 1.56 GCUPS while our method achieved 1.85 GCUPS using SSE4, respectively 2.83 GCUPS using AVX2, to align the PacBio-Sim data set, which was 1.2 to 1.8 times faster than ksw2. The results of comparing against Parasail are shown in Fig. 3.20

As can be seen in Fig. 3.20 our approach outperforms Parasail with all SIMD ISAs on the SKX. Moreover, our generalised alignment scheduler scales very well with the number of threads reaching a peak performance of 217 GCUPS on the SKX, which is 3 times faster than the best result of Parasail (68 GCUPS). Using the same SIMD ISA SeqAn is roughly twice as fast as Parasail.

[5] `https://github.com/lh3/ksw2`, accessed 21th March 2018

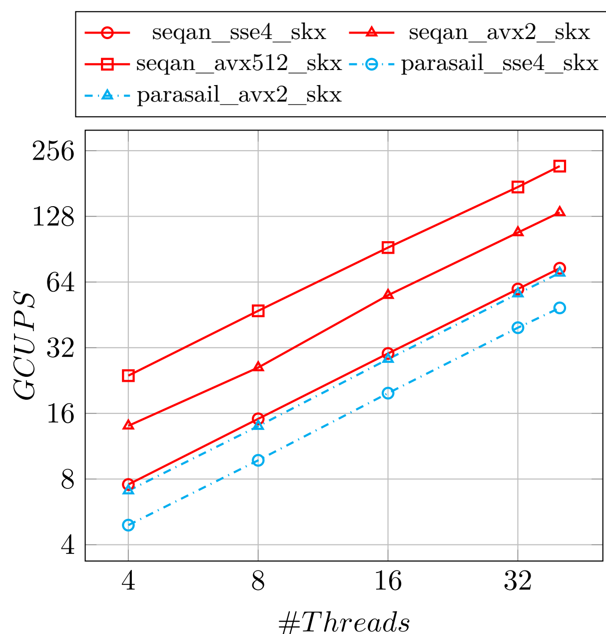


Figure 3.20: Performance and scalability comparison between our method and Parasail when aligning simulated PacBio data on SKX.

In addition, we used a real data set obtained from the the bam file hg002_gr37_chr22.bam [6] for the genome HG002 of the Ashkenazim trio [Zook et al., 2016]. In the benchmark we realigned the PacBio reads with their corresponding reference region to simulate a typical alignment step in a read mapping process. The real data set contained 277 598 sequences with the smallest sequence having a length of 42 bases and the longest 61 989 bases and in average a length of 8238 bases.

	SKX (t = 40)		
	time	GCUPS	speedup
seqan SSE4/8	403.76	65.60	1.6
seqan AVX2/16	218.83	121.05	2.8
seqan AVX512/32	137.86	192.14	4.4
parasail SSE4/4	607.58	43.60	1.0
parasail AVX2/8	400.35	66.16	1.5
parasail sat	686.29	38.60	0.9

Table 3.8: Comparison of our generalised dynamic task-graph alignment with Parasail on SKX using the PacBio-Real data set and 40 threads. The speedup factor is based on Parasails execution time using SSE4 with 4 lanes.

Table 3.8 shows the peak performance for the alignment of the real PacBio data set on the SKX. There are many more smaller sequences in the data set, such that the optimal performance was reached with a smaller tile size. We measured the best performance using a tile size of 1500 (192 GCUPS). The best performance of Parasail was obtained by using

[6] `ftp://ftp-trace.ncbi.nih.gov/giab/ftp/data/AshkenazimTrio/HG002_NA24385_son/PacBio_MtSinai_NIST/MtSinai_blasr_bam_GRCh37/hg002_gr37_22.bam`, accessed 14th November 2017

the *scan* algorithm with fixed 32-bit operand types using AVX2 (66 GCUPS). This is over 4 times slower than our approach using AVX512 with 32 lanes.

3.6.5 Evaluation of novel improvements

In this section, we present the preliminary results of our latest improvements in the vectorised interleaved execution of bulk alignments. To evaluate these improvements, we updated our benchmark suite to compare our previous implementation with the new one. All experiments were conducted on the ICX processor, which supports the additional AVX512VBMI ISA subset that we utilised to enhance the performance of alignments configured with a matrix substitution cost function. It is important to note that for these experiments, CPU frequency scaling could not be disabled due to limited access to the provided systems. Furthermore, we focused only on single-thread performance to evaluate the changes made to the vectorised execution.

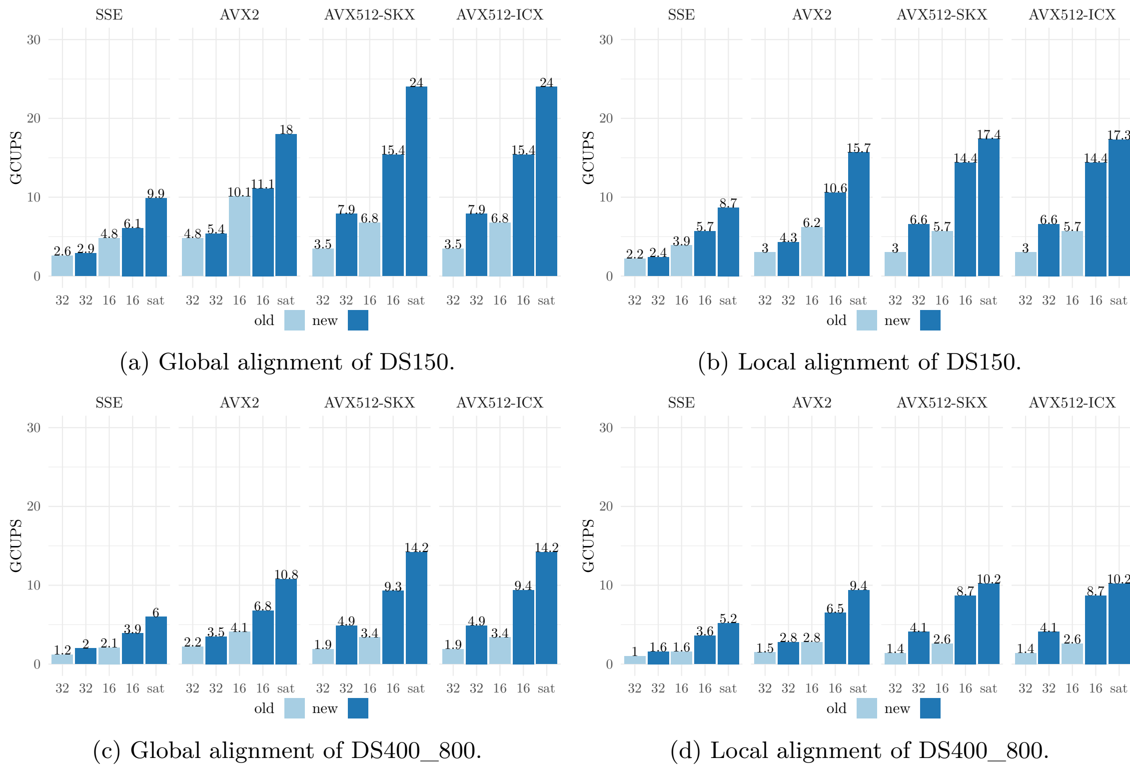
In our evaluation, we used four test datasets: DS150, DS400_800, AS500, and ASUniProt. DS150 and DS400_800 contain simulated DNA sequences. DS150 consists of 1000 alignment instances with each sequence having a length of 150 bases, while in DS400_800, the lengths are uniformly distributed between 400 and 800 bases. AS500 is a set of 1000 alignment instances from simulated amino acid sequences, with a length of 500 residues. ASUniProt contains the first 1000 sequences from the latest release of the UniProt database (UniProt, 2023).

For aligning the DNA sequences, we used the same unitary scoring scheme as described in the previous experiments. For the amino acid sequences, we employed the Blosum62 substitution matrix with the same affine gap costs. For each dataset, we computed global and local alignments using both fixed 32-bit and 16-bit score types. Additionally, if applicable, we evaluated the new saturation mode (sat). Each experiment was repeated 10 times, and the mean runtime of all runs was taken as the final result for computing the GCUPS (Giga Cell Updates Per Second). Correspondingly, a higher value indicates better performance.

Unitary costs

Figure 3.21 depicts the GCUPS achieved in the computation of affine alignments with unitary substitution costs. Analysing the two charts at the top, namely Fig. 3.21a and Fig. 3.21b, we can observe that even in the optimal case where all sequences have the same length and no additional masking for the regular ends of the individual DP matrices is performed, our new implementation is 10 to 30% faster for SSE and AVX2 when using the same score type.

These results can be mainly attributed to our low-level enhancements aimed at increasing instruction-level parallelism and optimising cache locality. Notably, in the case of AVX512, our new implementation outperforms the old one by a factor of 2 to 2.5. However, it is worth mentioning that while our new implementation scales as expected, the old implementation experiences a decline in performance. This drop seems to be related to the small dataset used, which computes only 1000 AIs to execute the benchmarking suite in a reasonable time. When we increase the dataset size to one million AIs, we could obtain comparable results that align with our previous observations for SSE and AVX2. Nonetheless, these



(a) Global alignment of DS150.

(b) Local alignment of DS150.

(c) Global alignment of DS400_800.

(d) Local alignment of DS400_800.

Figure 3.21: Performance comparison between old and new vectorisation when computing global and local alignments with unitary substitution costs. The labels of the bars below the x-axis reflect the used score type: 32 bit integer, 16 bit integer, and sat for 32 bit integer with our new saturation mode.

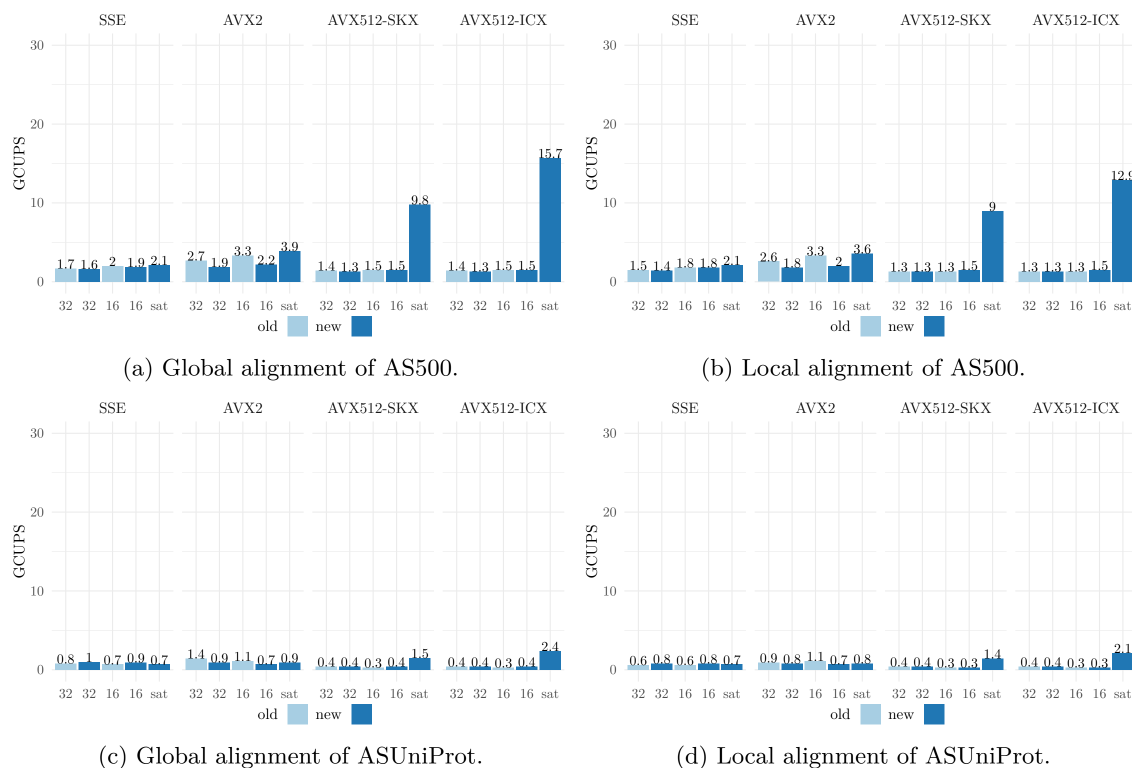
findings indicate that our new implementation can handle smaller datasets more efficiently, particularly as the width of the SIMD registers increases.

The saturation mode yielded the most significant improvement in performance. In the case of global alignments, we achieved a peak performance of 24 GCUPS, while for local alignments, the peak performance reached 17.4 GCUPS. Notably, the saturated AVX2 implementation even outperformed the 16-bit implementation using AVX512. It is worth mentioning that the relative performance gain for local alignments was lower compared to global alignments, which can be attributed to the higher complexity involved in switching between local and global tiles in the implementation.

Moving on to the heterogeneous datasets (Fig. 3.21c and Fig. 3.21d), we observed a significant improvement in performance due to our new strategy of computing interleaved collections of heterogeneous sequences. As indicated by the bar charts, our new implementation improved performance by 40% to over 100% for SSE and AVX2 compared to our previous version that utilised additional mask vectors. Once again, the peak performance was achieved with AVX512 using our novel saturation mode, resulting in an improvement from 9.3 GCUPS to 14 GCUPS for global alignments, and from 8.7 GCUPS to 10.2 GCUPS for local alignments.

Note there is no performance difference between AVX512-SKX and AVX512-ICX. That is, because for the DNA sequences we did not need the extra AVX512VBMI SIMD ISA.

Matrix costs



(a) Global alignment of AS500.

(b) Local alignment of AS500.

(c) Global alignment of ASUniProt.

(d) Local alignment of ASUniProt.

Figure 3.22: Performance comparison between old and new vectorisation when computing global and local alignments with matrix substitution costs. The labels of the bars below the x-axis reflect the used score type: 32 bit integer, 16 bit integer, and sat for 32 bit integer with our new saturation mode.

Our general solution using inter-sequence vectorisation performed exceptionally well for alignments using unitary substitution costs. However, when matrix substitution costs were used, severe performance issues arose due to the expensive gather instructions required to load scores from the scalar substitution cost matrix in each cell. Consequently, our solution was less efficient for computing alignments with amino acid sequences, for instance. To address this, we implemented a new generalised approach that replaced the gather instruction with more efficient permutation instructions.

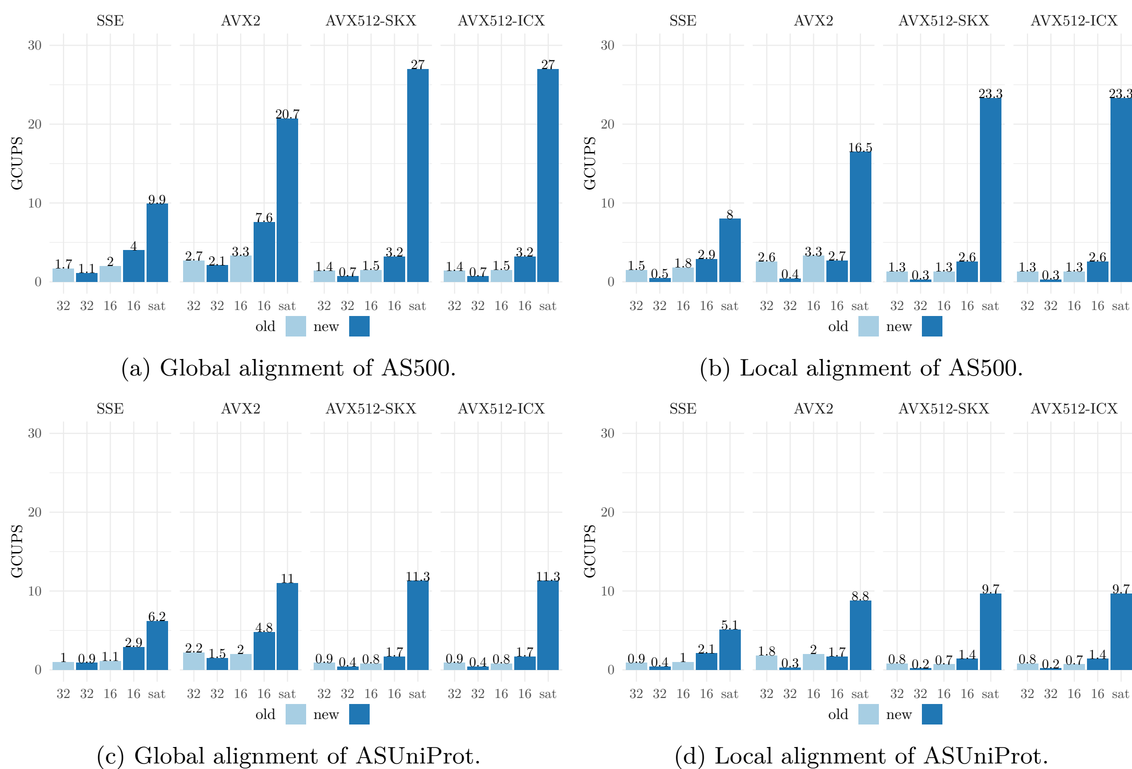
In our current version, this mode is implemented for alignments that can be computed with 8-bit score types and is applicable only to our saturated execution mode. To fully utilise this strategy, the AVX512VBMI instruction set (available on ICX) is required, as it provides the necessary cross-lane permutation instructions for 1-byte operand types. If AVX512VBMI is not available, but AVX512BW is (e.g. on SKX), we emulate the permutation using the cross-lane permutation instructions for 2-byte operand types, requiring four calls of the permutation operation instead of just two, along with additional packing and unpacking instructions.

The effects of these changes can be observed in Fig. 3.22a and Fig. 3.22b. The peak performance improved from 3.3 GCUPS (old 16-bit implementation using AVX2) to 9.8

GCUPS for AVX512-SKX and 15.7 GCUPS for AVX512-ICX in the global alignment case. Similar improvements were observed for local alignments, with a peak performance of 9 GCUPS for AVX512-SKX and 12.9 GCUPS for AVX512-ICX. The ASUniProt example also showed a significant performance improvement compared to our previous implementations.

However, it is important to note that the ASUniProt dataset consists of highly heterogeneous sequences, which limits the expected performance gains of the inter-sequence vectorisation layout. Nevertheless, when considering the results obtained for homogeneous sequence collections, it suggests that integrating our new SIMD implementation with the dynamic alignment scheduler may further improve performance for this dataset. The alignment scheduler would break down individual alignment instances into smaller tiles, allowing for mitigating the effects of such highly heterogeneous sequence collections.

Matrix costs in profile mode



(a) Global alignment of AS500.

(b) Local alignment of AS500.

(c) Global alignment of ASUniProt.

(d) Local alignment of ASUniProt.

Figure 3.23: Performance comparison between old and new vectorisation when computing global and local alignments with matrix substitution costs using the profile execution mode. The labels of the bars below the x-axis reflect the used score type: 32 bit integer, 16 bit integer, and sat for 32 bit integer with our new saturation mode.

We have extended our vectorised DP implementation to include a dedicated search mode, which is commonly used for aligning amino acid sequences. In this mode, we focus on aligning a single query sequence against a database of sequences. This specialisation of the alignment problem allows for efficient utilisation of precomputed score profiles, as described by Rognes [2011], who implemented it for the SSE SIMD ISA.

To analyse this profile execution mode, we selected the first sequence from the AS500 and ASUniProt datasets and aligned it against all the remaining sequences in the respective datasets. The results for computing global and local alignments are shown in Figure 3.23.

From the charts, it is evident that the performance improves significantly when using fixed 16-bit integers for the score type. However, with the saturation mode, the performance increases dramatically, reaching a peak performance of 27 GCUPS for global alignment and 23.3 GCUPS for local alignment on the AS500 dataset. For the ASUniProt dataset, the peak performance is 11.3 GCUPS (global, AVX512) and 9.7 GCUPS (local, AVX512).

It is worth noting that we can run the profile mode with only the AVX512BW SIMD ISA, so we did not observe any difference between the ICX and SKX target configurations. However, the improvement between AVX512 and AVX2 appears to be limited in the ASUniProt experiment. Similar to our previous observations, this can be explained by the fact that the likelihood of having larger differences in sequence sizes, collected into the same interleaved DP matrix, increases as more sequences are added. Consequently, we expect that a tiled execution using the task-graph implementation described earlier can mitigate this effect.

3.7 Universal algorithm design

In this thesis, we have implemented and combined several different approaches to optimise the performance of DP algorithms used in a wide range of applications. Alongside these achievements, a major concern of our work was to provide a unified implementation for the various DP algorithms and optimisations that is intuitive to use for both library developers and users while maintaining high functionality.

The importance of this can be illustrated by earlier versions of the SeqAn library. In these versions, the same or very similar DP algorithms were implemented repeatedly in different modules of the library. This led to implementations that were tightly coupled with the objects and functions within the respective modules. For example, the global alignment using affine gap costs was implemented in the align module, the graph module, the seeds module, as well as in a coexisting seeds2 module and in the SplazerS application, with slight modifications to the underlying DP algorithm. There were versions that only computed the score and versions that additionally computed the alignment. Similar situations occurred with other implementations, such as the global alignment with linear gaps, the local alignment, and banded alignments, among others. In total, we found over 30 different implementations that utilised more or less the same algorithm at their core.

Clearly, maintaining and modernising the DP algorithms for all these applications became impossible in this state. It was challenging to provide coherent documentation and a user-friendly API, which ultimately led to significant frustrations for both library developers and users [Kahlert, 2015]. Achieving our goal of universally accelerating not just a single, but the entire class of DP algorithms became infeasible given the added complexity of providing correct and efficient implementations of the presented code-level optimisations.

Algorithm decomposition

To refactor the implementations of the DP algorithms, we began by dismantling the different DP implementations and identifying a list of recurring features present in the algorithms implemented in SeqAn. These features were then organised into independent attributes that could be combined to form a complete and functional DP algorithm. Furthermore, we classified these attributes into three main categories, each affecting a different aspect of the underlying DP algorithm's implementation.

Alignment attributes	Features controlling the type and appearance of the computed alignment.
Matrix attributes	Features controlling storage and layout of the underlying DP matrix.
Execution attributes	Features controlling the execution pattern of the DP algorithm.

In Table 3.9, we provide a list of attributes that we identified in existing alignment algorithms or implemented in this thesis within SeqAn. These attributes are grouped according to their respective categories. It should be noted that this list may not be exhaustive, as we solely analysed algorithms implemented in SeqAn and there may exist other features that can be applied to our model.

Overall, we identified six attributes that influence the outcome of the alignment itself. The first two attributes pertain to the scoring scheme, while the third and fourth attributes specify the allowable start and end points of the final alignment. The **report** attribute determines the desired content of the final outcome, and the **objective** attribute further refines which alignments should be considered for the final solution.

Within the matrix category, we have identified four independent attributes. The **memory layout** attribute determines how the cells of multiple DP matrices are arranged in memory. The natural layout, known as **consecutive**, processes cells of each DP matrix sequentially, whereas the **interleaved** memory layout considers cells from different DP matrices in an interleaved fashion. We provided detailed information about this execution pattern in Section 3.3.

The **nesting** attribute enables the use of a decomposition strategy for the DP matrix, grouping a subset of cells into an additional layer that can be computed independently. By default, we employ a planar nesting strategy where a single DP matrix encompasses all cells. However, in this thesis, we extended this strategy with the **tiling** feature (see Section 3.3.5).

The **storage** attribute determines whether the alignment uses linear or quadratic space, while the **perimeter** attribute specifies whether all cells of the matrix should be computed or only a subset defined by a k-band or a chain of seeds, around which a banded-chain [Brudno et al., 2003a] is formed. Notably, many of these attributes are internally selected based on the alignment and execution settings.

	Attributes	Features
Alignment	substitution cost	matrix, unitary, position specific
	gap cost	linear, affine, dynamic
	begin position	first row, first column, first row and column, first row or column, any
	end Position	last row, last column, last row and column, last row or column, any
	report	score, positions, transcript
	objective	best, cooptimal, k-best, x-drop
Matrix	memory layout	consecutive, **interleaved**
	nesting	planar, **tiled**
	storage	linear, quadratic
	perimeter	all, k-band, banded-chain
Exec.	recursion pattern	column major, **striped**, **saturated**, **task graph**
	execution	sequential, **parallel**, **vectorised**
	input	single, **bulk**, **search**

Table 3.9: List of alignment algorithm attributes and their possible features identified in old versions of SeqAn or implemented in this thesis (bold).

The execution of the DP matrix can be controlled using three attributes. The **recursion** pattern determines the order in which cells in the DP matrix are computed. By default, a **column major** order is used to traverse the cells. In this thesis, we extended the recursion pattern to include a **task graph** recursion pattern (explained in detail in Section 3.4.2), a **striped** recursion pattern (see Section 3.5), and a **saturated** execution pattern (see Section 3.3.5).

The **execution** attribute specifies how the alignments are executed. This can be done **sequentially**, as discussed earlier, or in **parallel**, **vectorised**, or vectorised using a **saturation** mode.

Lastly, the expected input can be specified by computing a single alignment instance, a **bulk** of independent alignment instances, or by requesting to **search** a single sequence in a bulk of database sequences.

Composable algoroithm configuration

Notably, the attributes presented are independent of each other and can be combined. For example, selecting a specific substitution cost model is unrelated to the start or end points of an alignment or how the cells of the DP matrix are computed.

Based on this observation, we designed and implemented a unified DP algorithm abstraction that incorporates the DP attributes through customisable policy classes. Each policy class represents a specific attribute listed in Table 3.9. During a dedicated configuration phase at compile time, the final executable DP algorithm is assembled by instantiating the unified DP algorithm abstraction with the selected policies.

This decentralised design choice offers several advantages over the previous purely functional-oriented design. Firstly, it allows us to create a configuration system that hides the combinatorial complexity of DP attributes, providing an intuitive and consistent public API. This is demonstrated in the code snippet in Listing 1, which shows individual configuration steps for requesting a global alignment using affine gap costs and unitary substitution costs for DNA sequences. Users pass the configuration object along with the input sequences to the central `align_pairwise` function. Internally, we map the selected user configurations to the most efficient policy implementations available. For example, if users only request the alignment score (as in the above example), we always choose an implementation that consumes memory linear to the input sequences in each alignment instance.

Furthermore, runtime decisions can be used to switch between policy implementations because the configured algorithm is type-erased. Instead of returning the DP algorithm directly, we provide a lazy input range over the alignment results. This means that the next result is computed only when the iterator is incremented within the loop.

Listing 3.7.1: API to configure and compute pairwise alignments

```
std::vector input{std::pair{"AGTGCTACG"_dna4, "ACGTGCGACTAG"_dna4},
                  std::pair{"AGTAGACTACG"_dna4, "ACGTACGACACG"_dna4},
                  std::pair{"AGTTACGAC"_dna4, "AGTAGCGATCG"_dna4}};

// Configure scoring scheme
auto scoring_cfg = scoring_scheme{nucleotide_scoring_scheme{match_score{5},
                                                            mismatch_score{-4}}} |
                   gap_cost_affine{open_score{-10}, extension_score{-1}};
// Configure method and output
auto method_cfg = method_global{} | output_score{};

// Execute lazily
for (auto const & res : align_pairwise(input, method_cfg | scoring_cfg))
    debug_stream << "Alignment score: " << res.score() << "\n";
```

Listing 1: Example configuration and computation of a global alignment in SeqAn3. For brevity reasons we omitted the namespaces of the SeqAn3 entities and includes.

Secondly, this design approach allows application developers to prototype, optimise, and test new features in isolation before integrating them into the primary library or combining them with existing features. This significantly reduces code bloat by avoiding unnecessary code duplication and improves the time-to-release for new features. For example, we were able to implement, test, and document the scoring scheme agnostic x-drop feature for the protein aligner LAMBDA [Hauswedell et al., 2014], as well as an adaptation of the affine gap cost function called dynamic gap costs [Urgese et al., 2014], in a matter of hours.

Lastly, this design choice makes it easier to maintain consistency, comprehensiveness, and stability of the algorithms, particularly in our case where we incorporate different acceleration strategies to leverage thread-level, data-level, and instruction-level parallelism in a scalable manner. The importance of this work was also recognised by the reviewers of the corresponding publication [Rahn et al., 2018], who noted that the described abstraction

> ... represents an absolute tour de force of engineering, design, and conceptualization ... A main strength of this work is that it brings together the state-of-the-art advancements in pairwise sequence alignment with a sane (perhaps even elegant) interface that provides the user access to the combinatorial array of different alignment options ...

4 Compression-accelerated pattern matching

The rapid advancements in genomic sequencing technology, as described in our introduction, have enabled the generation of vast amounts of sequencing data within short time frames. However, this presents new challenges for the analysis of such data, particularly in the field of computational sequence analysis. To keep pace with the ever-growing volume of sequencing data (compare with Fig. 1.1), it is crucial to modernise current algorithmic strategies.

In this chapter, we shift our attention to algorithms and data structures for efficient management of population-scale sequences. The chapter begins with a brief overview of the background of the related subject, followed by a summary of the foundational work that has been accomplished in this area.

Subsequent sections provide a detailed description of the approaches we have evaluated in this work. First we will give a formal description of our compact data representation based on referential sequence compression, along with related concepts and algorithmic extensions. Following this, we will present a universal approach to extend any online algorithms originally applied to single sequences to our data structure. By leveraging the compressed representation of the sequence data, the execution of these algorithms is significantly accelerated compared to processing sequences individually. Lastly, we provide an experimental evaluation of our proposed data structure and algorithms in the context of pattern matching.

4.1 Background

From a medical perspective, the precise identification of disease-causing abnormalities through genome analysis is a critical aspect of diagnosing and treating rare or severe diseases [Di Resta et al., 2018]. This process involves two key steps: *genome inference* and *variant analysis*.

In the initial step, the DNA sequence of a patient is determined and aligned with a reference sequence to identify variants present in the investigated genome. This step is commonly referred to as the *primary sequence analysis*.

In the subsequent step, the detected variants are compared against databases containing known genetic variations. This comparison aims at excluding irrelevant variations and narrowing down the results to potentially harmful variants. This step is known as the *secondary sequence analysis*.

The accuracy of the primary and secondary sequence analyses play a crucial role in successfully identifying disease-causing aberrations and informing medical decisions for the diagnosis and treatment of these patients.

4.1.1 DNA sequencing

The standard approach for determining the genome of a sample is *DNA sequencing*. In this process, the original DNA molecule extracted from the sample is fragmented into smaller pieces. These fragments are often amplified and then subjected to a sequencing reaction, where chemical, physical, or electrical signals are generated for each processed nucleotide. Following the sequencing phase, the emitted signals are used to determine the actual nucleotide bases of these fragments, called *reads*, which are then stored in a digital file listing all sequenced reads [Lee et al., 2016; Metzker, 2010; Slatko et al., 2018]. However, to obtain the complete genome sequence, the acquired reads need to be reassembled in a process known as de novo *assembly* [Li et al., 2012b]. This assembly step is algorithmically challenging due to various biological and technical factors.

One challenge arises from the presence of repetitive regions in eukaryotic sequences, which can lead to difficulties in determining the correct order and arrangement of the reads during assembly. Additionally, there may occur errors during the sequencing process, resulting in inaccuracies contained in the read data. These factors contribute to the complexity and time-consuming nature of the *de novo* assembly process [Holtgrewe, 2015; Pop and Salzberg, 2008; Wee et al., 2019].

4.1.2 Resequencing

As a result of significant advancements in sequencing technologies over the past 20 years, new sequencing methods known as NGS have become widely adopted, along with accompanying analytical methods [Bao et al., 2011]. NGS technologies offer the capability to perform a vast number of sequencing reactions in parallel, increasing throughput and reducing the cost per base sequenced. However, compared to Sanger sequencing, the reads obtained through NGS are shorter and often have higher error rates, making them less suitable for *de novo* assembly.

In light of these limitations, the concept of resequencing has emerged, where existing reference genomes serve as templates for the assembly process. In the primary analysis step, the short reads are aligned or mapped against a known *reference genome*. This process is also known as read mapping. By repeatedly sequencing the same sample, the average coverage of each sequence position can be increased to multiple reads. Variants in the sample genome can then be identified by analysing the aligned reads. The identified variants are subsequently classified and filtered during the secondary analysis.

With recent advancements in single-molecule sequencing, also known as long-read sequencing, there is an expectation of a growing number of eukaryotic reference sequences in the future. Long-read sequencing technologies can generate reads in the kilobase range, making them well-suited for de novo assemblies. However, these technologies generally have higher error rates and are more cost-intensive compared to NGS. Therefore, a common approach is to combine long-read sequencing with short-read sequencing, leveraging the short reads to correct potential errors in the long reads. This hybrid approach improves assembly quality while keeping costs relatively low by requiring less overall coverage.

4.1.3 Reference bias

Even in the ideal scenario where all errors resulting from technical or biological artifacts are eliminated, the results of the primary analysis can still be influenced by the choice of the reference sequence. This issue is not primarily related to the completeness of the reference sequence itself. In the current version GRCh38[1], almost the entire sequence is known, with only 875 remaining assembly gaps to fill and a total of 160 million unknown bases [Guo et al., 2017; Marx, 2013; Schneider et al., 2017].

Instead, the issue can be attributed to a phenomenon known as *reference bias*, which refers to the observation that the reference sequence fails to fully capture the allelic diversity present in the entire population under consideration [Ballouz et al., 2019; Rosenfeld et al., 2012]. As a result, reads from sample genomes that exhibit higher divergence from the reference sequence may be misaligned or not aligned at all, particularly if they come from a non-reference allele. In fact, there is a tendency for such reads to be more likely aligned to the reference alleles. Subsequent variant calling approaches may struggle to correctly identify rare variants or miss them altogether. This bias is then propagated to the secondary analysis step, where the bias of the primary analysis can lead to a misinterpretation of the data.

To evade or at least reduce the issue of reference bias, researchers are currently investigating data structures and algorithms that enable the inclusion of the entire genetic diversity present in the population during resequencing. The necessary genetic information can be obtained from variation databases collected in various sequencing projects such as the 1KGP. Depending on the objectives, these databases encompass the genetic diversity of several thousand genomes. Consequently, one of the primary challenges in designing suitable data structures is to store the vast amounts of data in a compact form without sacrificing the ability to analyse them efficiently. This has given rise to the research field known as *computational pangenomics*.

4.1.4 Computational pangenomics

First studies focusing on microbial and bacterial genomes from larger cohorts summarised the set of all known genes as the *pangenome*. In the original definition, the pangenome was characterised by a pair consisting of a *core genome* and an *alternate genome* [Medini et al., 2005; Rasko et al., 2008]. The core genome comprised genes shared among all strains of a microbial or bacterial family, while the alternate genome contained genes present only in a subset of these sequences. However, the authors of Marschall et al. [2018] used a much broader definition and referred to the pangenome as '... any collection of genomic sequences to be analysed jointly or to be used as a reference.' In the following we may also refer to these sequences as *haplotypes* to denote single chromosomal sequences of multiploid organisms.

Marschall et al. [2018] listed several core challenges for computational pangenomics, including *completeness*, *stability*, *comprehensibility*, and *efficiency* of pangenomes. While all of these challenges are important, completeness and stability are mainly influenced by external factors such as the availability of sufficient financial and technical resources. Comprehensibility

[1]The most recent assembly version GRCh38.p13 is available from `https://ftp.ncbi.nlm.nih.gov/genomes/all/GCA/000/001/405/GCA_000001405.28_GRCh38.p13`; accessed on May 25th, 2021

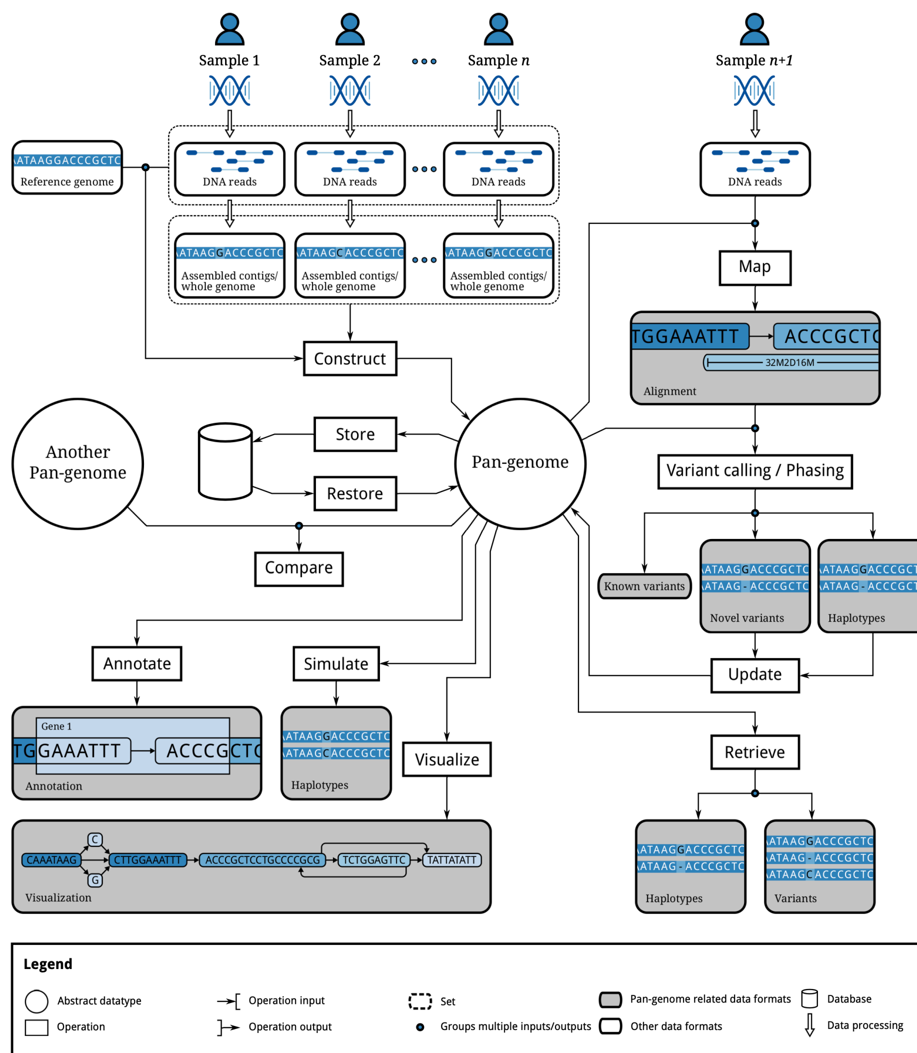


Figure 4.1: Illustration of operations to be supported by a pan-genome data structure (image and caption taken from Marschall et al. [2018]).

is a general biological problem based on the underlying complexity of the genomic traits modelled by a pangenome and can vary across different applications. Efficiency, on the other hand, primarily concerns the computational aspects of computational pangenomics.

In this thesis, we will specifically focus on the latter aspect. Our research is centred around the design of a compact pangenome data representation that reduces the required memory capacity for storing and loading pangenomes and can be used in standard bioinformatic sequence analysis routines.

According to Marschall et al. [2018], such a data structure should be efficiently constructible and dynamically updatable. Furthermore, it should provide a dedicated search interface, such as using it as a reference for aligning reads in a resequencing project. This also requires the support of a coordinate system that allows navigation within the pangenome structure and the unambiguous representation of pangenomic positions, enabling the assignment and

retrieval of auxiliary information to and from the pangenome, respectively. Supporting these features is a major requirement for pangenome models to be considered as a universal replacement for a single reference sequence in primary sequence analysis [Sherman and Salzberg, 2020]. Figure 4.1 provides an overview of the various operations demanded by a pangenome data structure.

4.2 Related work

In its simplest form, a pangenome can be represented as a set of linear sequences, as illustrated in Fig. 4.2. However, this data model is not applicable to pangenomes involving eukaryotic sequences due to their large genomic sizes. Instead, the common approach used by most pangenome representations is to store the sequences in a compressed format based on a *differential encoding* of the sequence variations present in the sequences. This method is effective because we can assume that sequences coming from the same population are quite similar to each other. In fact, analyses of short-read sequencing data suggest that the entire human pangenome, containing all auxiliary genetic information, will be approximately 10% larger than the GRCh38 human reference assembly [Sherman et al., 2019].

Graphical pangenome structures are commonly used to represent pangenomes as they model genomic variations within a given population through nodes in the graph and allow dynamic updates of the graph structure by adding or removing nodes and edges. These graphs can be augmented with additional support structures to establish a coordinate system or retrieve additional information regarding the underlying haplotypes, such as the original haplotype sequences. They can also be efficiently searched using an additional graph index structure [Durbin, 2014; Hickey et al., 2020; Novak et al., 2017; Sirén et al., 2020] and several tools have been developed to support sequence-to-graph alignment [Garrison et al., 2018; Jain et al., 2019; Kim et al., 2019; Rakocevic et al., 2019; Rautiainen et al., 2019].

In contrast, *reference-centric pangenomes* store the sequences in the form of a central reference sequence together with a set of alternate sequences encoding the genomic variations present in the encoded haplotype sequences. The general advantage of this approach is that, compared to graph-based pangenomes, most algorithms and applications working on linear sequence representations can be naturally extended to these schemes.

In the following, we will provide a brief summary of state-of-the-art methods and data structures, starting with the graphical pangenomes.

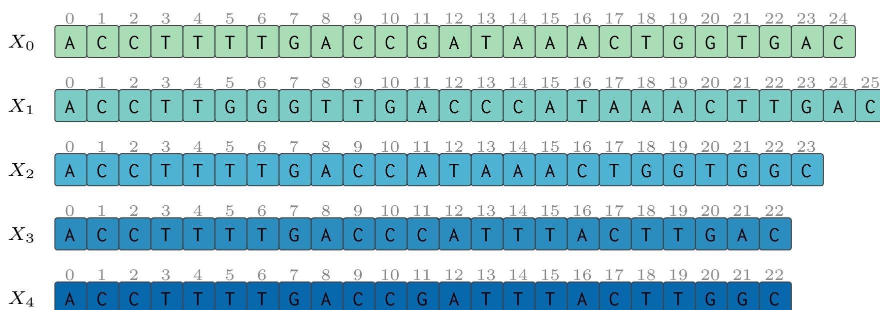


Figure 4.2: Pangenome $X = \{X_0, X_1, X_2, X_3, X_4\}$ consisting of 5 distinct plain sequences (different blue shades).

4.2.1 Graph-based pangenome structures

Graphical pangenomes are naturally well suited to model the biological complexity and diversity underlying the pangenomes due to their dynamic design and the availability of efficient graph algorithms. Typically, these models are based on node- or edge-labelled *sequence graphs*, where each label represents an alternate sequence present in one of the haplotypes of the modelled pangenome.

Alignment graph

The *alignment graph* constitutes one of the earliest sequence graph designs and was used to represent multiple sequence alignments, as shown in Fig. 4.3 [Hein, 1989; Lee et al., 2002; Rausch et al., 2008]. On a genome level, these graphs allow for the more efficient representation of complex genomic features, such as structural variations like insertions or rearrangements, by partitioning the sequences into segments.

An alignment graph was implemented and used as part of the segment-based MSA aligner within SeqAn [Rausch et al., 2008]. In this alignment graph, each haplotype itself is represented as a path of nodes, which are connected with directed edges and labelled with the corresponding segment of the original sequence. The path label, which is the concatenation of all node labels along the path, represents the original haplotype sequence. Furthermore, nodes from different haplotypes are connected via undirected edges if their segments are part of a collinear segment match.

There are other representations such as A-Bruijn graphs [Pevzner et al., 2004], Enredo graphs [Paten et al., 2008], or Cactus graphs [Paten et al., 2011] that can also be used to model multiple sequence alignments. However, these representations can be converted into each other, so we will not provide a detailed explanation here. Instead, we refer interested readers to the comprehensive review by Kehr et al. [2014], who described the operations for converting between different graph representations. These graph structures can be used in general to describe a pangenome as well. Herbig et al. [2012] showed that it is also possible to establish a global coordinate system based on the numbering of segments and an indexing of columns inside collinear aligned segments for multiple sequence alignments.

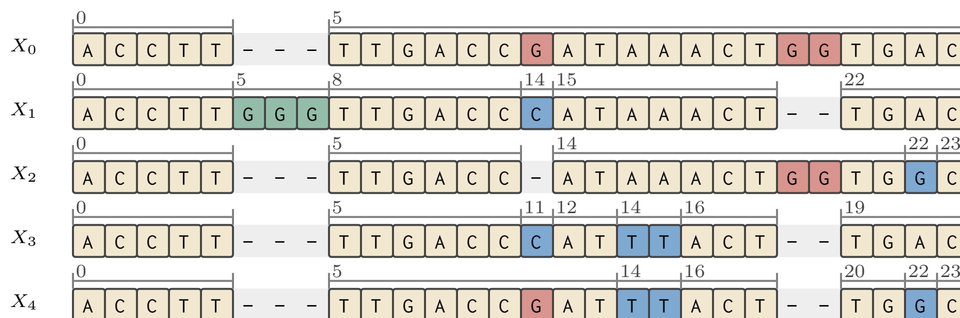


Figure 4.3: Multiple sequence alignment of sequences from Fig. 4.2

Genome graphs

A general extension of segment-based alignment graphs is the *genome graph*, which provides a space-efficient representation. In its basic form, a genome graph is an acyclic graph that condenses collinearly aligned segments into single nodes [Paten et al., 2017]. The space requirements for genome graphs can be further reduced by allowing cycles, which describe repetitive regions present in the sequences. Polymorphic sites in the pangenome are represented by multiple paths through the genome graph that begin and end in a shared head and tail node. These structures are also called *bubbles* [Paten et al., 2018].

To recover a particular sequence from a genome graph, it can be augmented with dedicated *haplotype paths*, as shown in Fig. 4.4. These paths describe the sequence of nodes in the genome graph, and the concatenation of their path labels represents the respective haplotype. In this case, a genome graph is also called a *variation graph* [Garrison et al., 2018], which is the central data structure around which the *VG*-library is built. The library comprises various tools to construct, serialise, index, search, or manipulate variation graphs. In order to impose some sort of coordinate system, Garrison et al. [2018] proposed augmenting nodes of the variation graph with relative sequence positions of the corresponding haplotypes covering a particular node. Another approach, proposed by Paten et al. [2018], included mapping a node to a reference position based on the topological order of the graph.

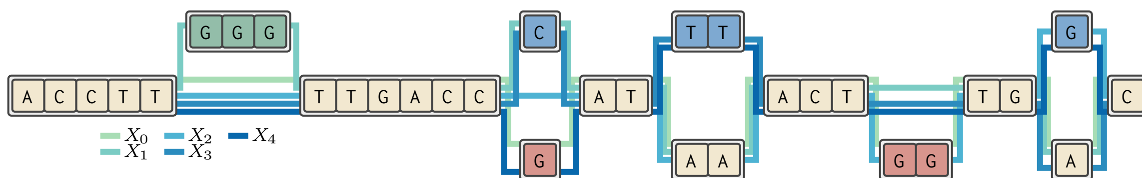


Figure 4.4: Variation graph with haplotype paths (edges with different blue shades) of the sequences from Fig. 4.2.

De Bruijn graphs

Another compact graphical representation of a pangenome can be derived from coloured de Bruijn *graphs* [Bolger et al., 2017; Holley and Melsted, 2020; Muggli et al., 2017; Turner et al., 2018]. Unlike segment-based alignment graphs and genome graphs, each node in a de Bruijn graph is labelled with a fixed-size sequence of length k, known as k-mers. To construct this graph, haplotypes are divided into consecutive overlapping k-mers, and nodes are connected with edges based on matching label prefixes and suffixes. By incorporating color information, the original haplotype information can be extracted from the graph as well.

4.2.2 Reference-centric pangenome models

In contrast to graphical pangenome representations where sequences are stored in no particular order, reference-centric pangenome models use a single reference sequence as a guide to encode the remaining haplotypes of the pangenome as sequence variants. This design allows reference-centric pangenomes to inherently support coordinate systems based

on the positions of the linear representation of the reference sequence. As a result, these models fit well into current sequence analysis pipelines that are built on the assumption of processing data with respect to a linear reference sequence.

VCF files

A classical example of this model is the *variant calling format* (VCF)[2], which is the standard file format for storing population-wide sequence variations. It stores in a tabular layout a list of all variants for a given set of samples, ordered by their genomic loci in the reference sequence. Each row describes the alternative sequence at a specific site and indicates which haplotype contains that particular alternative. If available, additional annotation data necessary to interpret the significance of variants in the clinical context may also be stored in each row. However, the primary focus of VCF and similar formats is on storing variant data, and other models are required to transform this data into efficient search data structures.

Context-aware alternate sequences

Alternatively, the pangenome can be stored as an extended list of plain sequences, where the first sequence is the reference sequence followed by alternate sequences along with their mappings to the reference [Church et al., 2011, 2015]. However, alignment programs are not specifically designed to handle variant data in this format. Typically, they treat alternate sequences as individual sequences and consider variants as repeats. This limitation has been addressed by existing aligners like BWA, which introduced a dedicated *alt-aware* mode to handle such cases [Li and Durbin, 2009]. However, as the size of the model increases, including the number of haplotypes and variants, this method becomes increasingly unsustainable.

Huang et al. [2013] demonstrated that classical read mapper, in this specific case BWA [Li and Durbin, 2009], could be extended to deal with population data. To achieve this, they padded alternate sequences with their local sequence context left and right of their genomic position inside of the reference sequence, and treated them as individual haplotypes. In addition, they used IUPAC characters to represent sites with single nucleotide polymorphisms as a single symbol inside the reference sequence. This way, they could use the same indexing strategy as the original mapper. However, the size of the context is application specific and depends on the length of the searched patterns. Thus, the index must be rebuild if the length of the reads changes due to a new sequencing protocol. On the other hand choosing the context too large would incur additional costs, as the same read will be mapped to multiple haplotypes although they represent the same location inside the pangenome.

Elastic-degenerate sequence

An adaption of the context-aware list model is the *elastic-degenerate sequence* model [Iliopoulos et al., 2016]. This model represents the pangenome as a sequence interleaved with arrays of alternative sequences, referred to as *elastic-degenerate symbols*. Each elastic-degenerate symbol represents the variations present at a specific location in the reference

[2] `https://github.com/samtools/hts-specs/blob/master/VCFv4.4.pdf`

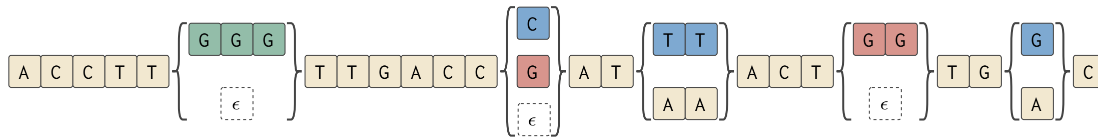


Figure 4.5: Pangenome represented as an elastic-degenerate sequence. Each elastic-degenerate symbol (braces) represents the alternative sequences present in the modelled pangenome ordered by their position with respect to a common reference sequence. Haplotype information are not captured in this model.

sequence. In the case that a segment of the reference sequence is shared by all haplotypes in the pangenome the corresponding sequence is represented as a plain sequence (see Fig. 4.5). Several pattern matching algorithms have been applied to this model to address the problem of finding exact and approximate occurrences [Bernardini et al., 2017, 2020; Cisłak et al., 2018; Grossi et al., 2017; Iliopoulos et al., 2016; Pissis and Retha, 2018].

Our contribution

In this thesis, we have developed and implemented a dynamic and scalable data structure for managing a large panel of similar sequences. We refer to this data structure as *referentially compressed multisequence* (RCMS). The core design of the RCMS is based on a reference-centric approach, which allows for a compact representation of the alternate sequences present in the underlying haplotypes. Additionally, we have adapted our structure to include a customisable graph-related search interface, yielding a hybrid design that combines the advantages of both pangenome representations.

In addition to space efficiency and support for dynamic updates, a major focus of our work was the design of a customisable and highly integrable API to support the relevant pangenomic operations summarised in Fig. 4.1. This enables easy integration of third-party search algorithms, provided they can be invoked through a standardised interface that is natural for online algorithms.

Furthermore, we have developed a set of small utility tools for creating, manipulating, viewing, and searching a RCMS. This allows developers and experimentalists to seamlessly upgrade their existing bioinformatic analysis routines by incorporating our RCMS in a straightforward manner. The initial version of this work was published in Rahn et al. [2014] in the Journal of Computational Bioinformatics. In this thesis we will also explore various extensions that we implemented since then.

First, we will provide a formal description of the core data structures involved in the design of the RCMS and review important aspects of its implementation. We will also explain how to create an RCMS from a sequence variant database, how to update an existing RCMS by adding or removing entire sequences or by updating existing sequences, how to retrieve haplotype-specific information from the RCMS, and how to perform generic searches in the RCMS using a graph-based traversal. Finally, we will demonstrate the interoperability, usability, and efficiency of our structure and algorithms by evaluating various pattern matching use cases and implementing a prototypic read mapper.

4.3 Referentially compressed multisequence

In the following description of the methodologies, we assume that we are given a finite, non-empty set $X = \{X_0, X_1, \cdots, X_{N-1}\}$, where $X_h \in \Sigma^*$, consisting of N sequences. We can make the assumption that the sequences in X are highly similar to each other, based on the targeted usage scenario. For instance, in the case of human genomes, it is known that two sequences share more than 99% of their bases [Auton et al., 2015]. As a result, we can significantly reduce the memory required to store the entire input data by applying a *referential sequence compression* scheme that eliminates redundant sequence information.

4.3.1 Data representation

As reference sequence, we select arbitrarily any sequence from the initial set X and store it entirely in the RCMS. Let $r \in X$ denote this reference sequence. We then encode each of the remaining sequences, referred to as the *target sequences*, as a set of *sequence differences* to r, storing only these differences in the RCMS. Essentially, a sequence difference describes how a segment of the reference sequence is modified by a corresponding segment of the target sequence. As the reference sequence is already stored in the RCMS, we can represent the source segment using an ordered pair of its starting and ending positions within it. We refer to this pair as the *breakpoint* of the sequence difference.

Definition 4.3.1 (Breakpoint)**.** We represent a breakpoint b in r as an ordered pair $(i, j) \in \mathbb{N}_0 \times \mathbb{N}_0$, where $0 \leq i \leq j \leq |r|$, and define the special fields $low(b) = i$ and $high(b) = j$ to represent the low and high end position of b. We further use the term *breakends* to refer to these end positions.

Definition 4.3.2 (Breakpoint span)**.** Given a breakpoint b, we define the *breakpoint span*, denoted by *span*, as the distance between the high breakend and the low breakend of b, i.e. $span(b) = high(b) - low(b)$.

Definition 4.3.3 (Sequence difference)**.** Let b be a breakpoint in r and $a \in \Sigma^*$ be some sequence. We model a sequence difference as an ordered pair $d = (b, a)$, with fields $bpt(d) = b$ and $alt(d) = a$. We call the latter field the *alternate sequence* of d.

For conciseness reasons, we also write in the following $low(d) = low(bpt(d))$ to access the low breakend, $high(d) = high(bpt(d))$ to access the high breakend, and $span(d) = span(bpt(d))$ to get the breakpoint span of the breakpoint associated with a sequence difference d.

The proposed encoding suffices to handle all three fundamental variant types occurring in genomic comparisons, i.e. insertion, deletion, and replacement. That is, given a sequence difference d, we classify d as

- an insertion, if and only if $span(d) = 0$ and $alt(d) \neq \epsilon$;
- a deletion, if and only if $span(d) > 0$ and $alt(d) = \epsilon$; and
- a replacement, if and only if $span(d) > 0$ and $alt(d) \neq \epsilon$.

In the case of an insertion, we observe that no symbol is removed from the reference sequence, which is reflected by setting both breakends of the breakpoint to the same reference position. However, the alternate sequence must contain at least one symbol. Conversely, in the case of a deletion, the high breakend of the breakpoint must be strictly greater than the low breakend, while the alternate sequence should be empty. Finally, a sequence difference represents a replacement if the high breakend is strictly greater than the low breakend, and the alternate sequence is not empty.

Encoding a single target sequence

As mentioned previously, the primary objective of the RCMS is to achieve a space-efficient representation of the input sequences in X using referential compression, while also enabling efficient access to the encoded sequences. In this section, we introduce the fundamental concepts of encoding an entire target sequence through a set of sequence differences. The following section will elaborate on the extension of this approach to encode multiple sequences.

Let $r \in X$ be the reference sequence, we define

$$\mathcal{B}_r = \{b \in \mathbb{N}_0 \times \mathbb{N}_0 \mid 0 \leq \mathit{low}(b) \leq \mathit{high}(b) \leq |r|\}, \tag{4.1}$$

to denote the set containing all possible breakpoints that can exist for r. Furthermore, we define Δ_r as the set encompassing all possible sequence differences applicable to r as:

$$\Delta_r = \{(b, a) \in \mathcal{B}_r \times \Sigma^* \mid a \neq r[\mathit{low}(b)..\mathit{high}(b)) \text{ if and only if } \mathit{low}(b) < \mathit{high}(b)\}. \tag{4.2}$$

Following this definition, we enforce that every element $d \in \Delta_r$ replaces any subrange of r with an alternate sequence a that deviates in at least one position from the corresponding non-empty reference infix represented by the associated breakpoint b. Note that if the associated breakpoint span is 0, a can be chosen arbitrarily.

Moreover, we define a strict total order on the elements of Δ_r. We say that $d_i < d_j$, where $d_i, d_j \in \Delta_r$, if and only if at least one of the following three expressions is true:

$$\mathit{low}(d_i) < \mathit{low}(d_j) \tag{4.3}$$

$$\mathit{low}(d_i) = \mathit{low}(d_j) \wedge \mathit{high}(d_i) < \mathit{high}(d_j) \tag{4.4}$$

$$\mathit{low}(d_i) = \mathit{low}(d_j) \wedge \mathit{high}(d_i) = \mathit{high}(d_j) \wedge \mathit{alt}(d_i) <_{\text{lex}} \mathit{alt}(d_j) \tag{4.5}$$

To put it differently, we sort the sequence differences based on their low breakends. In cases where two sequence differences have the same low breakend, we additionally sort them based on their high breakends. Finally, if two sequence differences share the same breakpoint, we sort them lexicographically (see Definition 2.2.4) based on their alternate sequences.

Now, let $D_h \subset \Delta_r$ denote some subset of Δ_r. More specifically, we say that D_h represents a *proper differential encoding* of the target sequence X_h based on the reference sequence r if and only if all sequence differences listed in D_h are *collinear*, and if the original sequence can be decoded without losing any information.

First, we consider the property of collinearity. Given two sequence differences $d_i \in \Delta_r$ and $d_j \in \Delta_r$, where without loss of generality, we assume that $d_i < d_j$, we call these two variants

collinear if and only if $high(d_i) \leq low(d_j)$. In other words, if two sequence differences are not collinear, their breakpoints must overlap in at least one position, which would introduce ambiguities in the representation of the encoded sequence.

The second property ensures that each sequence difference is a proper difference, meaning that if we replace the reference segments indicated by the listed breakpoints in D_h with their associated alternate sequences, we will retrieve the original target sequence X_h. To achieve this, we use a linear decoding algorithm `decode`, as listed in Algorithm 4.1, which can recover the original sequence by taking the reference sequence r and the related differential encoding D_h as input. Therefore, we have $X_h = \texttt{decode}(r,\ D_h)$.

Algorithm 4.1: `decode`

Input: r, D

```
1 x ← ε
2 k ← 0
3 foreach d ∈ D do
4     x ← x + r[k..low(d)) + alt(d)
5     k ← high(d)
6 x ← x + r[k..|r|)
7 return x
```

The `decode` algorithm, shown in Algorithm 4.1, begins by initialising an empty sequence x and setting the initial reference position k to 0. It then iterates over the ordered set of collinear sequence differences. In each iteration (Lines 3 to 5), the algorithm retrieves the low breakend of the current sequence difference d. This low breakend represents the end position of the adjacent shared reference segment immediately preceding the alternate sequence in the encoded target sequence. The algorithm appends this segment, followed by the alternate sequence, to the end of the current sequence x. It then sets the start position of the next reference infix to the high breakend of that sequence difference. After processing all sequence differences in order, the algorithm appends the remaining suffix of r to x. Finally, the fully decoded sequence x is returned.

In Section 4.3.4, we will introduce a data structure built on this idea that provides a regular sequence interface for a referentially compressed sequence, enabling fast random access to its elements.

Encoding mulitple target sequences

Given the high similarity between the sequences in the original pangenome and the frequent occurrence of shared variants among the population, it is reasonable to assume that many of the encoded sequence variants in the compact pangenome are shared across individual sequences. Therefore, it is desirable to store identical sequence variants only once and use an auxiliary structure to indicate the membership of a specific encoded variant in a particular accessory sequence. To capture this concept, we introduce the notion of *covered sequence differences*.

Definition 4.3.4 (Covered sequence difference)**.** Let $H_X = [0..|X|)$ denote the index set for the haplotypes in X, such that there exists an enumeration $f : H_X \rightarrow X$, where $f(h) = X_h$. We obtain a covered sequence difference by augmenting a sequence difference object d with the field $cov(d) \subset H_X$ and call this field the *haplotype coverage* of d. We further use the term *l-covered sequence difference* to denote a sequence difference d, whose coverage $cov(d)$ has a cardinality of l, i.e. $l = |cov(d)|$.

Assuming that we are given a proper encoding D_h for a particular sequence $X_h \in X$, then by Definition 4.3.4 each sequence difference is implicitly 1-covered, such that for all $d \in D_h$ there exists a singleton coverage with $cov(d) = \{h\}$.

With this in mind, we can construct a *generalised encoding* subsuming all individual proper encodings of the given input sequences, where identical sequence differences present in two or more encoded sequences are merged into a covered sequence difference. Let $\mathcal{D}$ denote such a generalised encoding over the individual encodings D_h for all sequences $X_h \in X$, we formally define this set as:

$$\mathcal{D} = \{(d, H) \in \varDelta_r \times \mathcal{H}_X \mid h \in H \text{ if and only if } d \in D_h\}, \tag{4.6}$$

where $\mathcal{H}_X = \mathcal{P}(H_X)$ denotes the power set over H_X and H represents the merged haplotype coverages for a sequence difference d present in multiple encodings.

In contrast to the proper encoding of a single sequence, the generalised set of sequence differences allows two or more sequence differences to have overlapping breakpoints. Nevertheless, the set is still well defined as we can assure through the associated coverages that sequence differences with overlapping breakpoints are covered by distinct sequences.

Theorem 4.3.1. Given two sequence differences $d_i, d_j \in \mathcal{D}$, where without loss of generalisation $low(d_i) \leq high(d_j) \leq high(d_i)$, then the intersection of their associated coverages must be the empty set, i.e. $cov(d_i) \cap cov(d_j) = \emptyset$.

Proof. If we assume that for these two sequence differences with overlapping breakpoints the intersection of their coverages yields a non-empty set, then by Eq. (4.6) there must exist a sequence $X_h \in X$, whose proper differential encoding D_h contains both sequence differences, i.e. $d_i, d_j \in D_h$. However, this would also mean that d_i and d_j in D_h are not collinear which contradicts our definition of a proper differential encoding. □

As a consequence of this, and because there cannot exist two distinct elements, whose breakpoints and alternate sequences are identical but have different coverages, we can impose a total ordering on the elements of $\mathcal{D}$. This also allows us to retrieve each of the original sequences by an adapted version of the `decode` algorithm shown in Algorithm 4.1.

Algorithm 4.2: `decode` from multiencoding

Input: r, $\mathcal{D}$, h

```
1 x ← ϵ
2 k ← 0
3 foreach d ∈ D do
4     if h ∈ cov(d) then
5         x ← x + r[k..low(d)) + alt(d)
6         k ← high(d)
7 x ← x + r[k..|r|)
8 return x
```

4.3.2 Implementation details

There are two general design goals that we considered for the implementation of the RCMS. On the one hand, the data structure should store the sequence differences as compact as possible, while on the other hand, it should provide fast access to the sequences in order to apply various operations as shown in Fig. 4.1 on them. Thus, we aimed for a solution that provides not only a compact but also an operational in-memory data representation without the need to decompress the data before using it. Looking at the results of various population studies, it becomes apparent that the majority of variants present in the underlying genomic datasets consist of *single nucleotide variants* (SNVs) (e.g. approximately 94% in the 1KGP as shown in Table 4.3). Correspondingly, we optimised the design of the main class so that it efficiently stores and accesses these overrepresented variants. In the following, we first present the abstract idea of our design to represent the generalised encoding in a space-efficient form and subsequently discuss different realisations of the data representation that provide different benefits depending on the actual use case.

The central RCMS, denoted by $\mathfrak{R}$, consists of three fields, namely the selected reference sequence r, a primary *breakend dictionary*, denoted by BRK, and a secondary *InDel dictionary*, denoted by IND. In general, the breakend dictionary uses as key the breakends of all breakpoints of the sequence differences present in the generalised encoding of a given pangenome and each key, i.e. breakend, is associated with the haplotype coverage. This representation alone, however, is not sufficient to fully encode a sequence difference because it misses the alternate sequence information. To handle this, we use two different strategies to encode SNVs and InDels in $\mathfrak{R}$.

SNV encoding

Overall, our target applications are based on DNA sequences, and thus we know that the alternate sequence of each of these sequence differences consists of one of the four symbols in $\Sigma_{\text{DNA}} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, with their ranks given as $\mathtt{A} : 0, \mathtt{C} : 1, \mathtt{G} : 2, \mathtt{T} : 3$. Following this, we encode the alternate symbol of a particular SNV within the key of the breakend dictionary itself.

In this scheme, we split the key into two parts: a *code* and a *position*. Let $\mathfrak{R} = (r, BRK, IND)$ be an RCMS, and let $\delta \in \mathfrak{R}$ be an element representing an encoded sequence difference in $\mathfrak{R}$.

We use $code(\delta)$ to access the code and $pos(\delta)$ to refer to the corresponding breakend position. Correspondingly, we define the key as $key(\delta) = (code(\delta) \ll 29) \,|\, (pos(\delta) \,\&\, 1 \ll 29 - 1)$.

The code is represented by the three most significant bits (i.e. bits 29, 30, and 31) of the key, which itself is represented by a 32-bit unsigned integer type. We use the two lower code bits (i.e. bits 29 and 30) to store the rank of the alternate symbol of a SNV. The most significant bit (MSB) is reserved as a marker to indicate whether the respective breakend belongs to an SNV (in that case, the MSB is 0) or an InDel (in that case, the MSB is 1).

The position of the key, i.e. the low or high breakend of a breakpoint, is represented by the lower 29 bits (i.e. bits 0 to 28). For most genomic applications, this is sufficient since a whole genome is typically stored as a set of chromosomes or contigs, from which the largest one is typically smaller than the largest representable integer value with 29 bits (e.g. the human chromosome 1 has roughly 247 Mbp). We further reduced the memory consumption by storing only the low breakends of all SNVs and insertion variants since we can infer their high breakends solely from their types. For a deletion, however, we also add a key-value pair to the breakend dictionary corresponding to the high breakend mate.

InDel encoding

Now, we consider the representation of a sequence difference representing an InDel variant. Conveniently, we can reuse the two lower bits of the code to discriminate between the three types of breakends. We define an InDel code alphabet $\Sigma_{\text{code}} = \{\texttt{100}, \texttt{101}, \texttt{110}\}$, where `100` represents the low breakend of a deletion, `101` an insertion breakend, and `110` the high breakend of a deletion. Furthermore, we can interpret these codes as numbers, i.e. $\texttt{100} = 4$, $\texttt{101} = 5$, and $\texttt{110} = 6$.

This numbering scheme provides us with the desired ordering of the breakends inside $\mathfrak{R}$.

Definition 4.3.5. Given two elements $\delta_i, \delta_j \in \mathfrak{R}$, we say $\delta_i < \delta_j$ if and only if

$$pos(\delta_i) < pos(\delta_j) \vee pos(\delta_i) = pos(\delta_j) \wedge code(\delta_i) > code(\delta_j).$$

Following this, the elements are naturally sorted by their breakends in ascending order and if two or more elements have the same position, they are ordered as follows: first, high breakends of deletions ending in this position (note that the high breakend points to one position behind the end); second, breakends of insertions; third, low breakends of deletions beginning at this position; and lastly SNVs.

To store the relevant information of insertion and deletion variants, we introduced the secondary InDel dictionary *IND*. Conceptually, the elements of this structure are linked by the elements of the breakend dictionary *BRK*, whose code value is greater than or equal to 4, i.e. $code(\delta) \geq 4$. Correspondingly, the size of *IND* corresponds one-to-one to the number of insertions plus two times the number of deletions present in the generalised encoding. Each element in *IND* is itself implemented as a union type. This union type represents

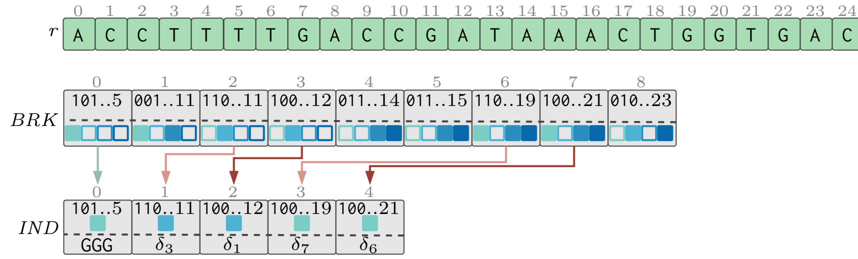


Figure 4.6: Conceptual representation of the sequences in Fig. 4.2 in form of a RCMS structure. The keys in BRK and IND are shown above the dashed lines in the grey boxes and the values below. The type of the variant is encoded in the first three bits of each key. The position in r is represented by the integer after '..'. Elements at index 2, 4, 5, and 8 in BRK store the SNVs, whose key codes encode the symbols `C` (001), `T` (011), `T` (011), and `G` (010) respectively. The coverages are represented in form of small squares. The colours match the sequences in Fig. 4.2. A filled square indicates that this sequence covers the linked sequence difference. The value of the insertion (index 0 in BRK) is stored in the first entry of IND (green arrow). The low breakends of the deletions (index 1 and 6 in BRK) map to index 1 and 3 (light red arrows) in IND, whose values store a pointer to their high breakend mate in BRK (index 3 and 7) and vice versa. Keys in IND store in addition one element from the haplotype coverage.

either the alternate sequence of an insertion or a handle to the deletion breakend mate (the other end of the associated breakpoint) in the breakend dictionary.

To identify the correct elements in IND, we augment the original key $key(\delta)$ with the haplotype coverage associated with the breakend. This guarantees that there can not be two identical insertions or deletions with the same breakpoint and overlapping coverage, according to the invariants of a generalised pangenome encoding. In practice, we only choose a representative coverage value to obtain a unique key, e.g. by choosing the first index from the respective haplotype coverage object. This is sufficient, because by definition we know that no two overlapping variants can be shared by the same encoded haplotype. Figure 4.6 illustrates the layout of an RCMS encoding the toy pangenome shown in Fig. 4.2.

Accessor fields

For convenience, we define the following accessor fields to recover the information of a sequence difference encoded as $\delta \in \mathfrak{R}$. To access the coverage, we use $cov(\delta) = val(BRK[\delta])$, where $BRK[\delta]$ refers to the key-value pair of the breakend dictionary corresponding to δ and val accesses the mapped value. The alternate sequence can be retrieved using the following function:

$$alt(\delta) = \begin{cases} \Sigma_{\text{DNA}}[code(\delta)] & \text{if and only if } code(\delta) < 4, \\ val(IND[\delta]) & \text{if and only if } code(\delta) = \texttt{101}, \\ \epsilon & \text{otherwise.} \end{cases} \tag{4.7}$$

Again, $IND[\delta]$ refers to the key-value pair of the InDel dictionary corresponding to δ. In the case where δ represents a SNV, we convert the code, which is equal to the rank of the stored SNV value, from the key of the associated breakend dictionary entry $BRK[\delta]$ to the corresponding symbol. If δ is an insertion, we retrieve the value of the entry in the secondary InDel dictionary IND that corresponds to δ. In all other cases, the breakend

represented by δ must belong to a deletion, and therefore, the alternate sequence is the empty sequence ϵ.

Next, we define the functions to access the associated low and high breakend values. For this we use a helper function, denoted by *mate*, which we define as:

$$mate(\delta) = \begin{cases} val(IND[\delta]) & \text{if and only if } code(\delta) = \texttt{100} \text{ or } code(\delta) = \texttt{110}, \\ \delta & \text{otherwise.} \end{cases} \quad (4.8)$$

That is, *mate* returns the element in $\mathfrak{R}$ that corresponds to the high or low breakend mate in the case that δ is the low or high breakend of a deletion, respectively. In all other cases δ is returned. Having this, we retrieve the low breakend of a $\delta \in \mathfrak{R}$ by:

$$low(\delta) = \begin{cases} pos(mate(\delta)) & \text{if and only if } code(\delta) = \texttt{110}, \\ pos(\delta) & \text{otherwise,} \end{cases} \quad (4.9)$$

and we access the high breakend by:

$$high(\delta) = \begin{cases} pos(mate(\delta)) & \text{if and only if } code(\delta) = \texttt{100}, \\ pos(\delta) + 1 & \text{if and only if } code(\delta) < 4, \\ pos(\delta) & \text{otherwise.} \end{cases} \quad (4.10)$$

The low breakend corresponds to the position of the breakend dictionary key, unless it represents the high breakend of a deletion. In that case, we return the position of the element representing its low breakend mate. As mentioned earlier, we only store the low breakends for SNV and insertion variants, as we can easily retrieve their high breakend. For a SNV, the high breakend is $pos(\delta) + 1$, and for an insertion, it is $pos(\delta)$. To access the high breakend of a deletion breakend, we return the current key position if it already represents the high deletion breakend, or the position of its high breakend mate if δ represents the low breakend of a deletion. Later, we will explore how this information can be used to enable a graph-based traversal over $\mathfrak{R}$.

Customisation

We have implemented several customisation points that allow users to customise the data representation of the RCMS and control its space and runtime behaviour for different use cases. For instance, if 29 bits are not sufficient to represent the entire index space of the underlying reference sequence, users can instantiate the breakend multimap with a 64-bit unsigned integer type. Another way to customise the data representation of the RCMS is by providing different dictionary strategies. In this thesis, we implemented two general approaches.

Customising dictionary strategy

The first approach is suitable for read-only scenarios where we assume that the dictionaries are not modified. In these situations, we provide a dictionary strategy that stores the

keys and values of the breakend dictionary in two separate linear buffers (e.g. using `std::vector`). For the InDel dictionary, we offer a strategy based on a hashmap (e.g. using `std::unordered_map`). With this approach, we can perform binary search to search for elements in the breakend dictionary. The coverage value can be accessed by using the offset of the key in the linear buffer to locate the corresponding position in the coverage buffer. Additionally, if needed, we can access the corresponding InDel element in constant time.

However, updating this dictionary by inserting or erasing sequence differences would have a linear runtime complexity proportional to the number of elements in the breakend dictionary and InDel dictionary. To address this, we introduced a second dictionary strategy that utilises a dynamically updatable data structure, such as `std::multimap`, which supports constant-time insertion and deletion operations. This strategy is advantageous in situations where the RCMS is frequently updated at random positions.

Table 4.1 compares the runtime complexities of the dictionary strategies used in static and dynamic scenarios.

	find	**random insert/erase**	**append**
static	$\mathcal{O}(\log n_\mathrm{B})$	$\mathcal{O}(\log n_\mathrm{B} + n_\mathrm{B} + n_\mathrm{I})$	$\mathcal{O}(1)$
dynamic	$\mathcal{O}(\log n_\mathrm{B} + \log n_\mathrm{I})$	$\mathcal{O}(\log n_\mathrm{B} + \log n_\mathrm{I})$	$\mathcal{O}(1)$

Table 4.1: Comparison of run time complexities between dictionary strategies used in static and dynamic use cases, with $n_\mathrm{B} = |BRK|$ and $n_\mathrm{I} = |IND|$.

Finding an element in the static case requires a binary search over the key buffer of *BRK*, while accessing any of the mapped values can be achieved in $\mathcal{O}(1)$ using a `std::hashmap`. In addition, inserting and erasing at random positions takes, in the worst case, two times the size of *BRK*, i.e. to insert the key and the coverage value. The breakend mates of deletions in *IND*, which are represented as indices to the corresponding element in *BRK*, may need to be updated (e.g. when inserting elements into the breakend of an existing deletion), requiring an additional linear scan over the hash table.

In the dynamic case, finding an element can be done in logarithmic time. However, to find an InDel entry, we also need to perform a logarithmic search step in the InDel dictionary. Inserting an element has a complexity that is logarithmic in the size of *BRK* and *IND* to find the insertion positions, while inserting the data itself can be done in amortised constant time. Because iterators to elements in ordered maps are not invalidated after inserting or erasing elements, we do not need to update the handles to the deletion breakend mates in *IND*.

If we only append to the end of the dictionary, we can benefit from amortised constant insertion time in both dictionary strategies. However, based on our primary assumption that SNVs are overrepresented in the targeted use cases, we assume that $|IND| \ll |BRK|$, such that the terms related to the InDel dictionary can be neglected in practice.

4.3.3 Construction

We offer two options to construct a RCMS. The first option generates a RCMS by parsing an already existing sequence variation database in the form of a VCF file. The second option

builds a RCMS incrementally by aligning a sequence to the reference sequence and merging the extracted sequence differences from the alignment with the existing RCMS instance $\mathfrak{R}$. In the following, we will first describe the process of parsing a VCF file, followed by the incremental construction of an RCMS $\mathfrak{R}$.

Parsing a VCF file

In many cases, genetic information about an entire population is already present in the form of a VCF file. Therefore, we have implemented a parser that takes a VCF file and the accompanying fasta file containing the reference sequences, and converts the information stored in the VCF file into sequence differences that are added to the RCMS.

It is important to note that the sequence differences in the VCF file are already sorted by chromosome and reference position according to the VCF specification[3]. Hence, the parser processes the file record by record, extracting the breakends, alternate sequences, and coverages for each VCF record. It then inserts the sequence difference at the end of the dictionary in the RCMS.

The number of sequences represented by the final RCMS, which is the maximum size of the difference coverages, depends on the number of samples and the number of haplotypes per sample. The total haplotype count is inferred from the genotype information of the first record of each chromosome stored in the VCF file. Whenever the parser encounters a record with a chromosome ID different from the previously processed record, a new RCMS instance is created, instantiated with the corresponding reference sequence loaded from the accompanying fasta file. Therefore, a single RCMS structure maintains the sequence differences of one chromosome or contig, and multiple RCMS structures are maintained within a proper composite structure.

At the time of writing this thesis, we only supported the parsing of InDels and SNVs. In the future, we intend to extend the implementation to also include complex structural variants. Additionally, the conversion process only includes records containing phased genotype information for multiploid organisms. This means that if the genotype information in the VCF file consists of multiple haplotypes, the haplotype covering the alternative can only be unambiguously identified if it can be assigned to the maternal or paternal chromosome in the case of biploid genomes. If the phasing information is not present for a particular variant, we skip the corresponding record and log the record information in a separate log file.

Other limitations that may arise include ambiguities in the presented variants, resulting in conflicting sequence differences, such as two or more variants with overlapping breakpoints and coverages. In such cases, the user can choose between two conflict resolution strategies that we have implemented. The first strategy skips conflicting variants similar to the variants with unphased genotype information. The second strategy removes only the conflicting haplotypes from the coverage of the newly inserted sequence difference, ensuring that the global invariants for the final RCMS are maintained. In both cases, the parser emits informative messages to the log file.

[3] `https://github.com/samtools/hts-specs/blob/master/VCFv4.4.pdf`

Incremental updates

There are use cases where an existing pangenome needs to be updated by adding new sequences, or where the pangenome is built from scratch using a set of plain sequences. To accommodate these scenarios, we have added the capability to incrementally add sequences to an existing RCMS. This is achieved by computing the pairwise sequence alignment between the target sequence and the underlying reference sequence of the RCMS, and converting the resulting alignment into a proper differential encoding for the new sequence.

To accomplish this, we have divided the procedure into two stages. In the first stage, we compute the pairwise sequence alignment between the target sequence and the reference sequence. In the second stage, we merge the differential encoding obtained from the alignment with the existing RCMS.

Compressing large-scale sequences

In general, we employ an optimised DP algorithm to compute a global pairwise sequence alignment between the target sequence and the reference sequence, as discussed in Chapter 3. However, due to the high similarity of the sequences, we have implemented an additional heuristic based on the sequence aligner LAGAN [Brudno et al., 2003b] to expedite the alignment process.

This approach incorporates an iterative seed-and-extend step before the alignment computation. During this step, we utilise a q-gram index (see Line 11) over the reference sequence to efficiently locate exact matching seeds between the target sequence and the reference sequence. These seeds are then merged into longer anchors based on specific merge criteria, which determine the maximum allowed distance between adjacent seeds in the DP matrix. Subsequently, we compute the highest scoring chain over all anchors [Gusfield, 1997].

In cases where the region between two adjacent anchors is too large, we repeat the seed-and-extend step after relaxing the merging criteria and using a smaller seed size. After a predefined number of iterations, we utilise the final anchor chain as a guide to compute the final alignment using a banded DP algorithm around the corresponding regions indicated by the chain.

It is worth noting that it is possible to use any other alignment algorithm to compute the alignment as well. This flexibility allows us to add a new sequence purely based on an alignment object, as demonstrated in the next section.

Merging sequence differences

After obtaining the pairwise sequence alignment between the target sequence and the reference sequence, we transform it into a suitable differential encoding and merge it with the existing RCMS instance. Recall that a pairwise alignment consists of two aligned sequences, where the original sequences are interspersed with special gap symbols ($-$) to indicate the presence of InDels in the alignment. Thus, for a target sequence $s \in \Sigma^*$ and

the reference sequence r from the RCMS $\mathfrak{R} = (r, BRK, IND)$, we are given an alignment $\mathbf{A} \in \mathcal{A}(r, s)$, with $\overline{r} = \mathbf{A}_{*,0}$ and $\overline{s} = \mathbf{A}_{*,1}$.

Firstly, we convert the alignment into an appropriate encoding, denoted as D_s. In the second step, we merge the elements of D_s into $\mathfrak{R}$.

Algorithm 4.3: `convert`

Input: $\overline{s}$, $\overline{r}$

Output: The proper encoding D_s of s

```
1  k ← 0
2  D_s ← ∅
3  d ← (0, 0, ϵ)
4  t ← { I   if and only if r̄_0 = −
         D   if and only if s̄_0 = −
         R   if and only if r̄_0 ≠ s̄_0
         M   otherwise
5  for i ← 0 to |r̄| do
6      if r̄[i] = − then // handle insertion
7          if t ≠ I then
8              t ← I
9              D_s ← D_s ∪ {d}
10             d ← (k, k, ϵ)
11         alt(d) ← alt(d) + s̄[i]
12     else
13         if s̄[i] = − then // handle deletion
14             if t ≠ D then
15                 t ← D
16                 D_s ← D_s ∪ {d}
17                 d ← (k, k, ϵ)
18             high(d) ← k + 1
19         else if s̄[i] ≠ r̄[i] then // handle mismatch
20             if t ≠ R then
21                 t ← R
22                 D_s ← D_s ∪ {d}
23                 d ← (k, k, ϵ)
24             high(d) ← k + 1
25             alt(d) ← alt(d) + s̄[i]
26         else // handle match
27             t ← M
28         k ← k + 1
29 D_s ← (D_s ∪ {d})
30 return D_s
```

The conversion algorithm is presented in Algorithm 4.3. It takes as input the aligned sequences $\overline{s}$ and $\overline{r}$. At the start, the algorithm initialises the current position k in r, the encoding D_s, the current sequence difference d, and a temporary tracker $t \in \{\mathrm{M}, \mathrm{R}, \mathrm{I}, \mathrm{D}\}$ to keep track of the last encountered edit operation – match (M), replacement (R), insertion

(I), and deletion (D).

Next, the algorithm iterates over the aligned sequences, where i denotes the current position in the aligned sequences. For each pair of aligned symbols $(\overline{s}_i, \overline{r}_i)$, the algorithm checks whether it represents an insertion (Lines 6 to 12), a deletion (Lines 13 to 18), a mismatch (Lines 19 to 25), or a match (Line 26 and 27).

If the current position represents an insertion, the algorithm checks whether it marks the start of a new insertion or extends an existing one. In the former case, it adds the current state of d to D_s, resets its *low* and *high* fields to the current reference position k, and resets its alternate sequence to an empty sequence. Additionally, it sets t to the insertion state I so that if the next aligned symbol is also an insertion, the current state of d can be updated instead of being reset. In either case, it appends the current symbol of the aligned sequence $\overline{s}_i$ (which, by definition of an alignment, must not be the space symbol) to the alternate sequence of the current sequence difference d.

If the current position represents a deletion, the algorithm first checks whether the previous position was also a deletion (i.e. $t = \mathrm{D}$). If this is not the case, then the current position marks the low breakend of a deletion. The current sequence difference d is appended to D_s before being reset, where the low breakend corresponds to the current reference position k. Additionally, t is set to the deletion state D to prevent resetting d if the deletion continues to the next position. After potentially resetting d, the high breakend value is set to the current reference position k plus one to mark the end of the deletion.

The same logic applies if the current position i represents a mismatch. After possibly updating D_s and resetting d, as done in the previous cases, the high breakend value represents the end of the replaced reference infix. Correspondingly, the current symbol at $\overline{s}_i$ is appended to the alternate sequence of d.

In all other cases, the pair of aligned symbols represents a match. In this case, only the marker t is updated accordingly, ensuring that the last state of d remains unchanged within a contiguous match segment. In addition to the insertion case, the current reference position, which progresses along the original reference sequence r, is incremented. Finally, the algorithm appends the last state of d to D_s and returns the final encoding of s.

Having converted the initial pairwise alignment to a proper encoding for the target sequence s, we can now merge this set with an existing RCMS $\mathfrak{R}$. Since both structures are basically sorted lists ordering their elements in an ascending order, this merging step can be done in time linear to the sum of their sizes.

4.3.4 Data retrieval

Accessing data from the compressed representation of the pangenome is another essential operation required by many applications, as it allows users to annotate the stored data with additional application-specific information. We have implemented three essential data retrieval operations: generating a *haplotype view*, as well as determining the *haplotype cover* and the *haplotype position* for a given reference position. These three operations serve as building blocks for more sophisticated data retrieval operations.

In the following, we discuss the details of these three operations, starting with the haplotype view, followed by the haplotype cover, and finally the haplotype position projection.

Haplotype view

The view operation allows the extraction of the original haplotype sequence from its encoded representation, as described in the extended `decode` algorithm (see Algorithm 4.2), without the need to duplicate all segments that are shared with the reference sequence. This operation provides a convenient read-only sequence view, allowing it to be accessed in the same way as its plain sequence representation.

To achieve this, we use an auxiliary data structure called a *journal*, which serves as a temporary buffer recording all sequence modification requests issued to the reference sequence. It is important to note that the reference sequence itself remains unmodified. We refer to a sequence with journaling support as a *journaled sequence* (JS).

Journal

The journal is the central structure of a JS. Conceptually, it can be thought of as a dictionary, where the keys represent positions in the JS, and the mapped values are pointers that refer to external sequences. It is important to note that the memory of these external sequences is not owned by the journal. In the following, we use the term *journal entry* to refer to an element in this dictionary.

Definition 4.3.6 (Journal entry)**.** Given some integer $p \in \mathbb{N}_0$ and a non-empty sequence $s \in \Sigma^* \setminus \{\epsilon\}$, then we define the journal entry e as the ordered pair (p, s).

In addition, we define the fields $pos(e) = p$ and $seq(e) = s$ to access the integer and the sequence value of e and define $|e| = |seq(e)|$. Then, given two journal entries e_i and e_j, we write $e_i < e_j$ if and only if $pos(e_i) + |e_i| \leq pos(e_j)$. Moreover, we say e_i and e_j are *connected* if and only if $e_i < e_j$ and $pos(e_i) + |e_i| = pos(e_j)$. In the presented context, this means that the sequence represented by e_j follows immediately after the sequence of e_i without introducing a gap in the final JS. Having this we can define the journal more specifically.

Definition 4.3.7 (Journal)**.** A *journal* $E = \{e_0, e_1, \ldots, e_{l-1}\}$ is a non-empty, strict totally ordered set over journal entries, where all of them, except the first one, are connected to their predecessor and it must hold that $pos(e_0) = 0$.

To put it differently, the journal entries stored inside a journal E, with $l = |E|$, form a continuous index space starting at 0 and ending at $pos(e_{l-1}) + |e_{l-1}|$. The concatenation of the associated sequences in the order given by E will then produce a sequence. Specifically we write,

$$\mathcal{J}(E) \coloneqq seq(e_0) + seq(e_1) + \ldots + seq(e_{l-1}), \tag{4.11}$$

where $\mathcal{J}(E)$ denotes the journaled sequence. This is illustrated in Fig. 4.7.

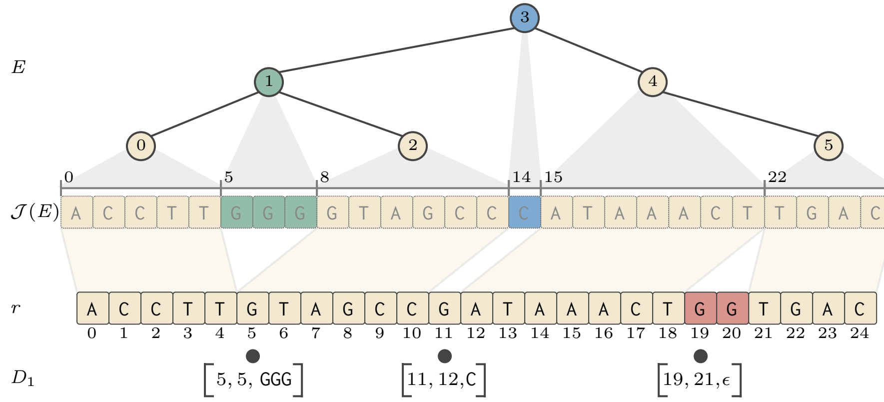


Figure 4.7: Visual representation of the encoded haplotype X_1 shown in Fig. 4.2 as a journaled sequence with $r = X_0$ being the source sequence. The journal E is shown at the top in form of a binary search tree consisting of 6 journal entries ordered by their position *pos* (black numbers above crossbars). Each journal entry covers either a segment from r (entries $0, 2, 4$ and 5) or an inserted sequence of a sequence variant (entries 1 and 3). A deletion is represented by two entries leaving a gap between the respective segments in r (entries 4 and 5).

Element access

As discussed previously, one of the main considerations in our design is the efficient access to any element within the JS without the need to allocate a new memory for the entire sequence. To accomplish this, we take advantage of the fact that the journal entries inside the journal are ordered based on their position offset within the represented JS. This allows us to access any element with an additional logarithmic time complexity. The algorithm `at` (listed in Algorithm 4.4) provides the retrieval of the p-th symbol from a given journal E.

Algorithm 4.4: `at`

Input: E, p

1 **if** $i \geq E[|E| - 1]$ **then**
2   **return** `NIL`
3 find $e \in E$, such that $pos(e) \leq p < pos(e) + |e|$
4 **return** $seq(e)[p - pos(e)]$

To access the element at position p of the JS $\mathcal{J}(E)$, the algorithm searches for the entry e that covers the requested position p (Line 3). The method of searching depends on the underlying dictionary strategy, which can involve a binary search using a sorted array or traversing a balanced binary search tree from its root. In either case, this step takes at most $\mathcal{O}(\log |E|)$ time. Based on the properties of a journal instance, we know that there can only exist one such element that covers the position, or the given position p exceeds the length of the represented JS. In the latter case, the algorithm returns a special state (`NIL`) indicating that the element was not found (Lines 1 and 2). If a matching journal entry is found, the algorithm determines the relative position of the queried position p within the found journal entry e and retrieves the corresponding symbol from the associated sequence $seq(e)$ (Line 4). In Fig. 4.8 the steps of the algorithm are exemplified.

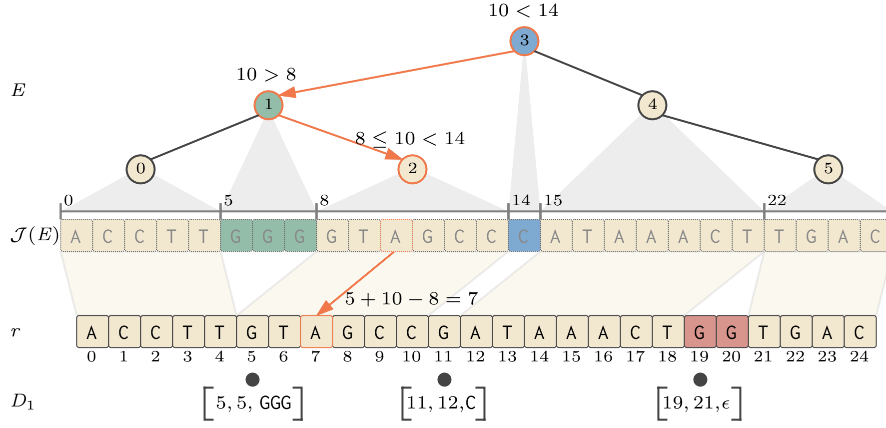


Figure 4.8: Example of accessing the 10-th base of the journaled sequence from Fig. 4.7. The binary search compares 10 with the entry positions finding e_2 in the end (orange arrows). This entry covers the infix $r_{5..11}$ starting at position 8 in the journaled sequence which corresponds to the 7-th position in r, i.e. $7 = 5 + 10 - 8$ (highlighted symbol).

Dynamic updates

In this section, we demonstrate how a JS can be efficiently constructed from an existing sequence, leveraging the dynamic nature of the journal structure. Let $r \in \Sigma$ be the reference sequence used as the basis for generating a target sequence $s \in \Sigma$ by modifying r accordingly. Instead of copying r, we aim to utilise a journal E to record the sequence modifications, such that s can be viewed as $\mathcal{J}(E)$. In other words, $\mathcal{J}(E)$ serves as a view of the target sequence s, using r as the basis.

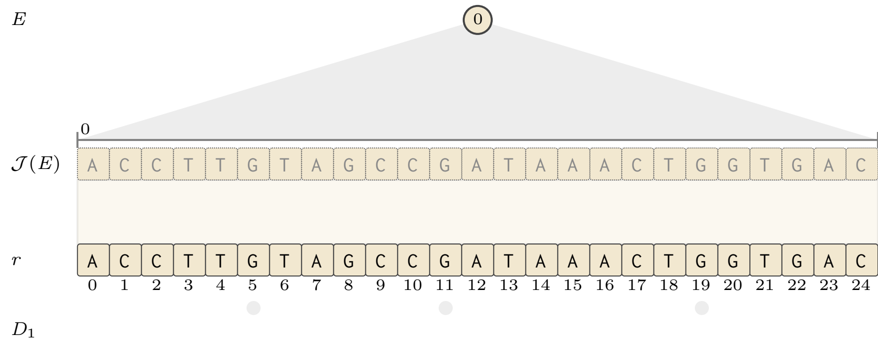
Initially, we initialise E with an entry that covers the entire reference sequence, as illustrated in Fig. 4.9a. At this stage, we have $r = \mathcal{J}(E)$. Next, we consider the cases where we insert, erase, and replace segments in r to obtain the target sequence s. A comprehensive example of all three operations is depicted in Fig. 4.9.

Algorithm 4.5: `insert`

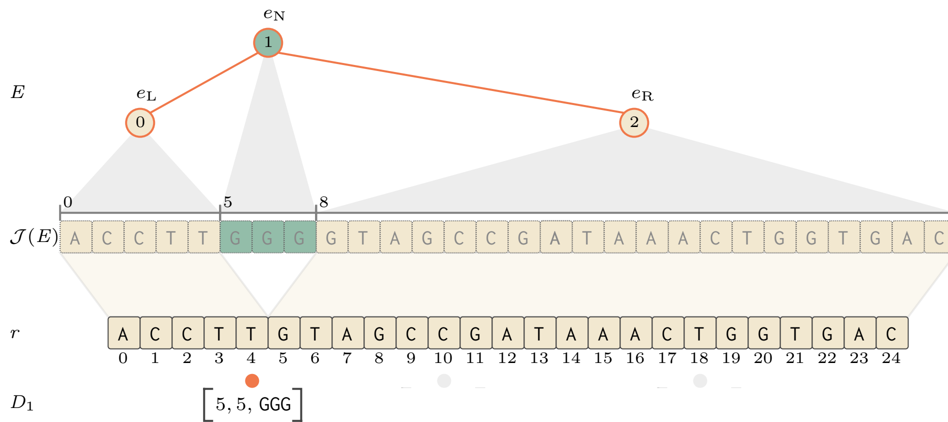
Input: E, $p \in [0..|\mathcal{J}(E)|]$, $x \in \Sigma^*$

1. find $e_L \in E$, such that $pos(e_L) \leq p < pos(e_L) + |e_L|$
2. $o \leftarrow p - pos(e_L)$
3. $e_R \leftarrow (p, seq(e_L)[o..|seq(e_L)|))$
4. $seq(e_L) \leftarrow seq(e_L)[0..o)$
5. $e_N \leftarrow (p, x)$
6. Insert e_N after e_L into E
7. Insert e_R after e_N into E
8. **for** $e_R \leq e \leq e_{|E|-1}$ **do**
9. $\quad pos(e) \leftarrow pos(e) + |x|$

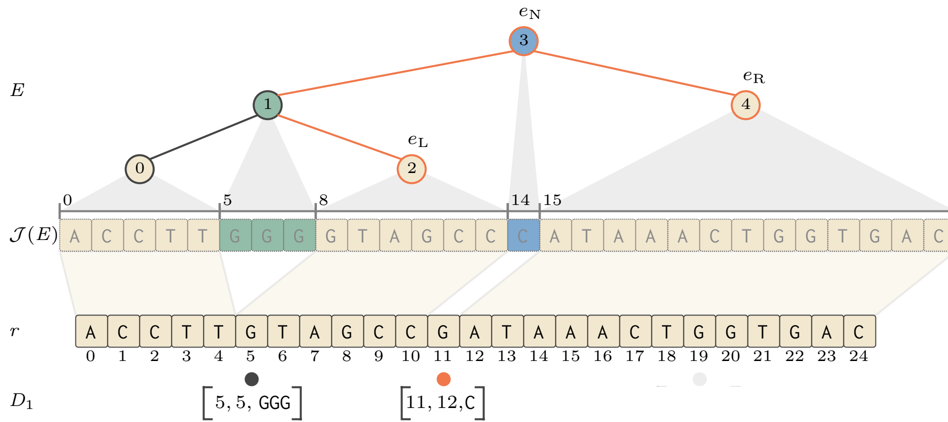
To insert a new segment $x \in \Sigma$ at position p into an existing journal E, we first search for the entry $e_L \in E$ that covers the insertion position p, as depicted in Line 1 of Algorithm 4.5. For the sake of brevity, we omit the additional check for a valid insertion position p and assume that it falls within the range of $[0..|\mathcal{J}(E)|]$. Note that $p = |\mathcal{J}(E)|$ indicates that x



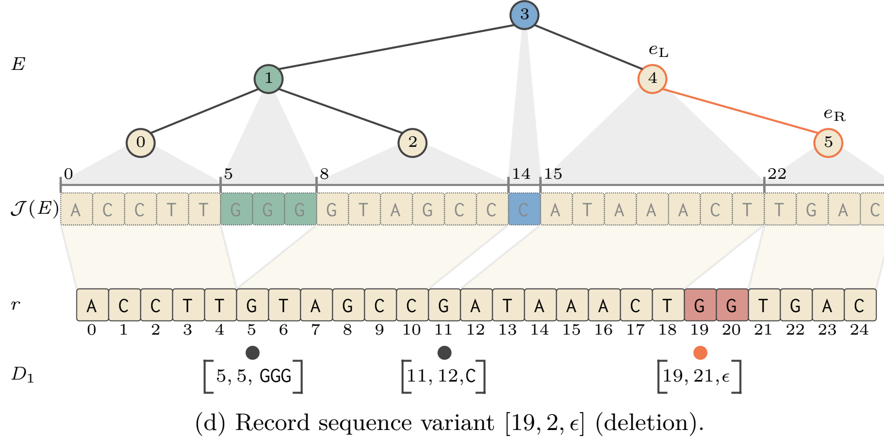
(a) Initial journaled sequence with $r = \mathcal{J}(E)$.



(b) Record sequence variant [5, 5, GGG] (insertion).



(c) Record sequence variant [11, 1, C] (replacement).



(d) Record sequence variant $[19, 2, \epsilon]$ (deletion).

Figure 4.9: Stepwise construction of the journaled sequence for X_1 shown in Fig. 4.7. Orange lines represent updated or added journal entries in the current step.

is inserted at the end of $\mathcal{J}(E)$. The found entry e_{L} represents the left prefix of the sequence immediately preceding the insertion point, denoted by the split position o computed in Line 2. Additionally, a new journal entry e_{R} is created, representing the corresponding suffix starting at o (Lines 3 and 4). This is illustrated for e_2 and e_4 in Fig. 4.9b. Subsequently, the algorithm creates a new entry e_{N} (e_3 in Fig. 4.9b) covering the new segment x, which starts at p in the modified JS. Finally, it resets the stored sequence of e_{L} and inserts e_{N} and e_{R} after e_{L} in the appropriate order. Depending on the chosen dictionary strategy, these elements can be inserted in amortised constant time, given that the correct insertion position is known from the initial search. Lastly, the algorithm iterates over all remaining entries that come after e_{R} and updates their positions by adding the length of the inserted segment x.

Algorithm 4.6: `erase`

Input: $E, p \in [0..|\mathcal{J}(E)| - c], c \in [1..|\mathcal{J}(E)|]$

1. find $e_{\mathrm{L}} \in E$, such that $pos(e_{\mathrm{L}}) \leq p < pos(e_{\mathrm{L}}) + |e_{\mathrm{L}}|$
2. find $e_{\mathrm{R}} \in E$, such that $pos(e_{\mathrm{R}}) \leq p + c < pos(e_{\mathrm{R}}) + |e_{\mathrm{R}}|$
3. $o_{\mathrm{L}} \leftarrow p - pos(e_{\mathrm{L}})$
4. $o_{\mathrm{R}} \leftarrow (p + c) - pos(e_{\mathrm{R}})$
5. **if** $e_{\mathrm{L}} = e_{\mathrm{R}}$ **then**
6. $\quad e_{\mathrm{R}} \leftarrow (p + c, seq(e_{\mathrm{L}})[o_{\mathrm{R}}..|seq(e_{\mathrm{L}})|))$
7. $\quad$ Insert e_{R} after e_{L} into E
8. **else**
9. $\quad seq(e_{\mathrm{R}}) \leftarrow seq(e_{\mathrm{R}})[o_{\mathrm{R}}..|seq(e_{\mathrm{R}})|)$
10. $seq(e_{\mathrm{L}}) \leftarrow seq(e_{\mathrm{L}})[0..o_{\mathrm{L}})$
11. **for** $e_{\mathrm{L}} < e < e_{\mathrm{R}}$ **do**
12. $\quad$ remove e from E
13. **for** $e_{\mathrm{R}} \leq e \leq e_{|E|-1}$ **do**
14. $\quad pos(e) \leftarrow pos(e) - c$

Algorithm 4.6 presents the pseudocode for erasing a contiguous range of size c from the

JS, starting at position p. Once again, we omit the sanity check at the beginning of the operation. In Lines 1 to 2, we search for the left entry e_L, which covers the begin position of the deletion in the JS, and the right entry e_R, which covers the corresponding end position (e_4 and e_5 in Fig. 4.9d). The algorithm computes the split positions for both journal entries, denoted as o_L and o_R, respectively. In other words, we want to erase the suffix starting at o_L in $seq(e_L)$, the prefix until o_R in $seq(e_R)$, and all entries between e_L and e_R. If the deleted segment is enclosed by a single entry, meaning $e_L = e_R$, a new journal entry is created, covering the corresponding suffix of e starting at o_R, and inserted right after e_L in the journal E (Lines 5 to 7). Otherwise, the sequence of e_R is set to the corresponding suffix that comes after the deletion (Lines 8 to 9). Afterward, the sequence of e_L is assigned the corresponding prefix until position o_L (Line 10). In the last step, the entries between e_L and e_R are removed (Lines 11 to 12), and the positions of the subsequent entries greater than or equal to e_R are updated by subtracting c from their positions.

Algorithm 4.7: `replace`

Input: E, $p \in [0..|\mathcal{J}(E)| - |x|]$, $x \in \Sigma^*$

```
1 erase(E, p, |x|)
2 insert(E, p, x)
```

The `replace` operation, as shown in Algorithm 4.7, can be implemented by first calling `erase` on the segment starting at p with a length of x, i.e. $|x|$, followed by inserting x at position p in the updated journaled sequence $\mathcal{J}$. While this approach does not change the complexity of the algorithm itself, we have implemented a more efficient strategy in practice that saves one logarithmic lookup, as we already know the insert position from the erase operation. The effects of replacing a symbol of the reference sequence are illustrated in Fig. 4.9c.

Implementation details

We have implemented the journal entry as a class consisting of an unsigned 32-bit integer and a handle to the associated sequence. Therefore, a journal entry is a lightweight data structure that only points to the memory location of the segment.

In the context of viewing an encoded haplotype inside a RCMS instance, the journal entry can either point to a memory address of the reference sequence r or to the stored alternate sequence of insertions. However, for SNVs, we needed a different approach since their alternate sequences are encoded within the keys of the breakend dictionary. To handle this, we implemented a special sequence handle type that internally uses a union. This union can represent either a contiguous chunk from an actual sequence memory address or the value of the encoded SNV in the form of a `std::array` of size one.

Table 4.2 compares the asymptotic run times of the three fundamental operations: `at`, `insert`, and `erase`, between a regular sequence s (storing elements contiguously in memory, e.g. a `std::vector`) and its journaled representation $\mathcal{J}(E) = s$.

Accessing an element in s can be done in constant time, while inserting or erasing an element will have a linear time complexity in the worst case. In comparison, operations on the

	at	insert	erase
s	$\mathcal{O}(1)$	$\mathcal{O}(\|s\|)$	$\mathcal{O}(\|x\|)$
$\mathcal{J}(E)$	$\mathcal{O}(\log \|E\|)$	$\mathcal{O}(\log \|E\| + \|E\|)$	$\mathcal{O}(\log \|E\| + \|E\|)$

Table 4.2: Comparison of run time complexities between a regular sequence s and its journaled representation $\mathcal{J}(E)$ of the three fundamental operations `at`, `insert`, and `erase`.

journaled sequence $\mathcal{J}(E)$ require an additional logarithmic step in the size of the journal E to find the corresponding entry. Furthermore, all journal entries to the right of an insertion or deletion need to be updated, which takes $\mathcal{O}(|E|)$ time.

However, it is important to note that the JS representation is specifically designed to view modifications between similar sequences. Therefore, we can assume that $|E| \ll |s|$, making the JS representation favourable in such scenarios.

Moreover, when viewing a haplotype from the RCMS, we can build the JS representation through a linear scan from left to right. This means that we only modify the last journal entry of E in each iteration, which can be done in amortised constant time.

Haplotype cover

So far, we demonstrated how to efficiently compress a large collection of sequences using a differential sequence encoding approach and presented a space-efficient strategy for viewing the original sequences in their compressed form. While this enables operations on individual compressed sequences without the need to allocate the entire sequence in memory, there are numerous scenarios in pangenomics where leveraging differential sequence compression can further reduce overall workload and significantly improve application performance.

One of these scenarios is the capability to identify all haplotypes covering a given position of the reference sequence.

Definition 4.3.8 (Haplotype cover)**.** Given a RCMS $\mathfrak{R} = (r, BRK, IND)$ for a pangenome X, we define the *haplotype cover* as the coverage $H_p \subseteq H_X$ at a given position $p \in [0..|r|)$.

The basic idea behind computing the haplotype cover for a given reference position p is to identify all sequence differences whose breakpoints overlap with position p and subtract them from H_X. If there are no overlapping breakpoints, it implies that all encoded sequences cover the specific position in the reference sequence, and therefore, $H_p = H_X$.

Computing the haplotype cover becomes straightforward when only insertions and SNVs are present in $\mathfrak{R}$. In this case, it is sufficient to find the lower bound δ_i and the upper bound δ_j in $\mathfrak{R}$ for a search position p such that $p \leq pos(\delta_i) < pos(\delta_j)$. The haplotype cover ${H_p}'$ can then be computed by subtracting the combined coverage of all covered sequence differences within the interval $[\delta_i..\delta_j)$:

$$ {H_p}' = H_X \setminus \bigcup_{k=i}^{j-1} cov(\delta_k). \tag{4.12} $$

In cases where deletions are also present in $\mathfrak{R}$, we need to consider sequence variants whose low breakends occur before p and high breakends occur after p. To address this, we augmented the breakend dictionary *BRK* with a *breakpoint tree*, which is an augmented interval tree [Cormen et al., 1990] built over the breakpoint spans of the deletion variants.

Formally, the breakpoint tree is a binary search tree denoted as $T = (V, E)$. A node $v \in V$ represents an ordered pair consisting of a sequence difference $\delta \in \mathfrak{R}$ and the *breakend supremum* $s \in \mathbb{N}_0$, which is the maximum high breakend value among all nodes in the subtree rooted in v. We also denote $low(v) = low(\delta)$, $high(v) = high(\delta)$, and $sup(v) = s$.

In practice, we represent the tree T as an array, where the size of the array is equal to the number of low deletion breakends (i.e. $code[\delta] = \texttt{100}$) present in $\mathfrak{R}$. The T can be constructed using Algorithm 4.8, which has a time complexity linear to the size of *IND*.

Algorithm 4.8: Construct breakpoint tree

Input: An RCMS $\mathfrak{R}$
Output: Breakpoint tree T

```
1 T ← ∅
2 foreach δ ∈ IND do
3     if code(δ) = 100 then // is low deletion breakend
4         v ← (δ, high(δ))
5         append v to T
6 findSupremum(T, 0, |T|-1)
7 return T
```

First, the algorithm initialises an empty breakpoint tree T. Then it iterates over the InDels stored in *IND*. Whenever the code of the key of the currently visited InDel difference δ indicates a low deletion breakend, the corresponding entry δ is appended to T together with its high breakend as the breakend supremum. After that, the construction algorithm calls a subroutine named `findSupremum` to recursively calculate and assign the breakend supremum for all inner nodes. The details of the `findSupremum` algorithm are provided in Algorithm 4.9.

Algorithm 4.9: `findSupremum`

Input: T, l, r

```
1 if l ≥ r then // is leaf!
2     return sup(T[l])
3 m ← ⌊(l+r)/2⌋
4 sup(T[m]) ← max { sup(T[m])
                    findSupremum(T, l, m-1)
                    findSupremum(T, m+1, r)
5 return sup(T[m])
```

The subroutine `findSupremum` is called with the initial state of T and the interval $[0..|T| - 1]$, which represents the currently visited subtree of T. In each iteration, the middle position m of the interval $[l..r]$ is computed (Line 3), corresponding to the root node of the current subtree.

The algorithm then determines the maximum value between its own associated breakpoint supremum and the breakpoint suprema of its left and right subtrees, and updates its own breakpoint supremum accordingly. To obtain the suprema of its subtrees, it recursively calls the `findSupremum` subroutine for the left subtree (i.e. `findSupremum(`T, l, m - 1`)`) and the right subtree (i.e. `findSupremum(`T, m + 1, r`)`). If `findSupremum` is called with $l \geq r$, it means a leaf node has been reached in T, and the recursion anchor is reached. At this point, the base supremum, which corresponds to the high breakend of the leaf node, is returned to the calling parent. After visiting all nodes in T, each inner node will store the breakend supremum of its subtrees. Since the position of the parent node is removed from the interval representing the left and right subtrees in each recursive invocation, every node in T is visited exactly once. Therefore, the runtime of this subroutine is linear, i.e. $\mathcal{O}(|T|)$, and the total construction runtime complexity is $\mathcal{O}(|IND| + |T|)$.

Algorithm 4.10: `search`

Input: Breakpoint tree T, reference position p
Output: Set of all overlapping deletions

```
1  R ← ∅
2  S ← ∅
3  push(S, (0, |T| − 1))
4  while S ≠ ∅ do
5      l, r ← pop(S)
6      m ← ⌊(l+r)/2⌋
7      if p > sup(T[m]) then // p larger than supremum
8          continue
9      push(S, (l, m − 1)) // Add left subinterval to search
10     if low(T[m]) ≤ p < high(T[m]) then // p overlaps deletion
11         R ← R ∪ {m}
12     if p < low(T[m]) then p not in right subinterval
13         continue
14     push(S, (m + 1, r)) // Add right subinterval to search
15 return R
```

After constructing the breakpoint tree T, we can query it to find all overlapping breakpoints in $\mathcal{O}(\min(|T|, o \log |T|))$, where o is the number of breakpoints that overlap a given position p. The corresponding search algorithm is shown in Algorithm 4.10. Initially, an empty result set R and a stack S are initialised. The stack tracks the visited nodes as closed intervals, similar to how the `findSupremum` subroutine was implemented. The initial interval corresponds to the root node of the tree representation. In each iteration, the current subinterval is popped from the stack, where l represents the left border and r represents the right border of the subinterval. The middle position m is computed to retrieve the element in T that corresponds to the root node of the current subtree.

If the searched position p is greater than the supremum *sup* of $T[m]$, it means that all breakpoints in the subtree rooted at m must end before p. In this case, the search can be stopped, and we continue with the next subtree tracked in S. Otherwise, we check if there is an overlapping breakpoint in the left subtree of m (Line 9). We also check if p is contained within the breakpoint of the current node, $T[m]$, and add m to the result set

R if this is true. Furthermore, we check if p is strictly less than the low breakend of the current breakpoint. If this condition is satisfied, we know from the construction of T that all breakpoints in the right subtree of an inner node must have a low breakend greater than or equal to the low breakend of the subtree root. Thus, there is no need to search the right subtree, and we continue with the next subtree in S. In the other case, we add the right subinterval to S and start searching the next subinterval at the top of S.

The algorithm terminates after visiting all nodes whose reference spans overlap position p. Having the final list of overlapping breakpoints R, and given that $cov(v) = cov(\delta)$, we can compute the final haplotype cover H_p as:

$$H_p = {H_p}' \setminus \bigcup_{v \in R} cov(v). \tag{4.13}$$

Similar to the `findSupremum` subroutine, by excluding the currently visited element from its left and right subinterval, we ensure that each element in T is visited at most once. However, since we can terminate the search of a subtree once it is clear that no breakpoint in the subtree can be added to the result set, we only need to traverse $\mathcal{O}(|R| \log |T|)$ nodes. Therefore, the overall runtime complexity is $\mathcal{O}(\min(|T|, |R| \log |T|))$.

Haplotype position

To answer the question of which encoded sequences cover a certain reference position, we added the capability to compute the *haplotype position*, which is a set storing the corresponding absolute positions in their decoded form.

Specifically, given a set of sequences X with $N = |X|$, and its differential encoding $\mathfrak{R} = (r, BRK, IND)$ for some reference sequence $r \in X$, we want to project a position $p \in [0..|r|)$ of the reference sequence to an ordered set of absolute sequence positions, denoted by P_p, which is defined as:

$$P_p \in [0..|X_0|) \times [0..|X_1|) \times \cdots \times [0..|X_{N-1}|),$$

where $P_p[h]$ represents the absolute position of the h-th sequence, X_h, in X that maps to position p in the reference sequence.

To answer such a query for a particular sequence $X_h \in X$, we first find the rightmost entry δ_i in $\mathfrak{R}$ such that $p \leq low(\delta_i)$. Then, we compute the cumulative sum over the *effective sizes*, which represent the number of symbols added or removed with respect to r, of all sequence differences covered by the h-th sequence up to and including δ_i.

Definition 4.3.9 (Effective size)**.** The effective size, denoted by $\sigma(\delta)$, of a sequence difference $\delta \in \mathfrak{R}$ is computed as the difference between the size of its alternate sequence and the breakpoint span, defined as follows:

$$\sigma(\delta) = |alt(\delta)| - (high(\delta) - low(\delta)).$$

Having this, we can compute the haplotype position by:

$$P_p[h] = p + \sum_{k=0}^{i-1} \begin{cases} \sigma(\delta_k) & \text{if and only if } h \in cov(\delta_k) \\ 0 & \text{otherwise,} \end{cases} \tag{4.14}$$

where we add to p the relative offset that results from adding up the effective sizes of all sequence differences whose low breakend comes before p in the reference sequence and where h is a member of the coverage.

Precomputing offsets

Notably, only sequence differences representing insertions and low deletion breakends need to be considered for the haplotype position, as these are the types that effectively change the projected position with respect to p. As a consequence, we introduced another augmenting data structure that, similar to supporting the computation of the haplotype cover, is solely bounded by the number of contained InDels. This structure is an offset matrix $[\mathbf{O}_{\imath,\jmath}]^{(M+1)\times N}$, where $\mathbf{O}_{\imath,\jmath} \in \mathbb{Z}$. The matrix has $M + 1$ columns and N rows, where M is equal to the sum of insertions and low deletion breakends present in $\mathfrak{R}$. Specifically, we only consider breakends with the key `101` (insertion) and `100` (low deletion breakend).

Let BRK' denote this subset of BRK, such that $M = |BRK'|$. A column $\mathbf{O}_{\imath,*}$ of $\mathbf{O}$ stores, in each row, the cumulative sum of the effective InDel sizes up to the $\imath$-th InDel breakend entry in BRK' according to Eq. (4.14). An additional column is used to initialise the offsets with 0. This matrix can be constructed by performing a linear scan over the sorted InDel dictionary, where for each $\imath \in [1..M]$:

$$\mathbf{O}_{\imath,\jmath} = \mathbf{O}_{\imath-1,\jmath} + \begin{cases} \sigma(\delta) & \text{if } \jmath \in BRK'[\imath - 1], \\ 0 & \text{otherwise.} \end{cases}$$

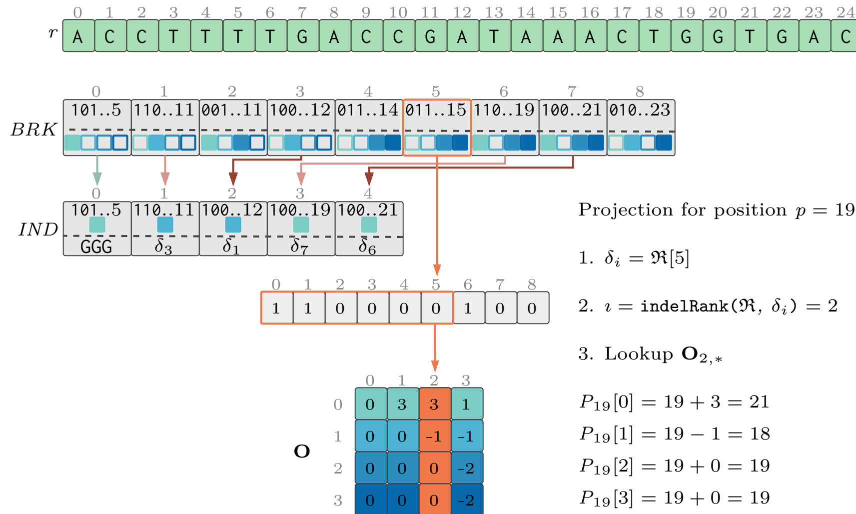


Figure 4.10: Example of computing the absolute positions for the encoded sequences in the RCMS from Fig. 4.6 that map to position 19 of r.

Rank support

Instead of maintaining a separate subset of BRK, we can use a rank dictionary over the breakends stored in BRK to count the number of InDels occurring before and up to the strict lower bound δ_i for a given reference position p in constant time. The resulting rank gives the corresponding column index for $\mathbf{O}$ (see Fig. 4.10). Specifically, we have

$$\texttt{indelRank}(\mathfrak{R},\, \delta_i) = \sum_{k=0}^{i-1} \begin{cases} 1 & \text{if and only if } code(\delta_i) = \texttt{100} \lor code(\delta_i) = \texttt{101} \\ 0 & \text{otherwise.} \end{cases}$$

Putting everything together, we can compute the P_p using the following steps:

1. Find the strict lower bound δ_i in $\mathfrak{R}$.

2. Determine the column index $\imath = \texttt{indelRank}(\mathfrak{R},\, \delta_i)$.

3. Read the corresponding offsets $P_p[\jmath] = \mathbf{O}_{\imath,*} + p$.

The first step takes $\mathcal{O}(\log |BRK|)$ time. The second step can be answered in constant time $\mathcal{O}(1)$ using a proper rank dictionary implementation.

In our current version, we used the rank implementations of the SDSL library [Gog et al., 2014]. Reporting the final absolute positions for all sequences can be done in $\mathcal{O}(N)$, thus the total worst-case runtime complexity is $\mathcal{O}(\log |BRK| + N)$. Since we only need to store the offsets for the indel variants, we need $\mathcal{O}(4N(M+1))$ bytes to create $\mathbf{O}$, plus approximately $\mathcal{O}(0.25|BRK|)$ additional bits to store the rank dictionary[4].

By combining all three retrieval operations to extract haplotype information, we are able to augment an RCMS object with additional functionality. For example, we can use these operations to view the sequence variations present in a specific pangenome region indicated by two reference positions or to extract auxiliary annotation data for a specific pangenomic region.

4.4 Dynamic traversal interface

In this section, we will explore how we can efficiently apply any window-based online algorithm on the RCMS without the need to decompress the individual sequences encoded within it. This capability has the potential to significantly speed up pangenome-based applications, as the overall workload can be greatly reduced thanks to the use of differential encoding.

[4] Based on the SDSL documentation: `http://simongog.github.io/assets/data/sdsl-cheatsheet.pdf`; accessed on 22.03.2023

4.4.1 Journaled sequence tree

We designed and implemented a tree-like traversal interface that allows navigation through the segments of the RCMS in a deterministic manner. Importantly, we have achieved this without the need to materialise an actual graph object. Instead, our tree is a virtual construct that provides a skeleton for realising the traversal. Additionally, we implemented various policies refining the traversal. These include the ability to prune subtrees of the sequence tree when it becomes evident that a concrete path is not represented by the encoded sequences or when further traversal into the subtree would not generate new relevant information for the applied online algorithm. As a result, we can dynamically apply various online algorithms with different window sizes without the need to rebuild the tree structure. A general illustration of our approach is depicted in Fig. 4.11.

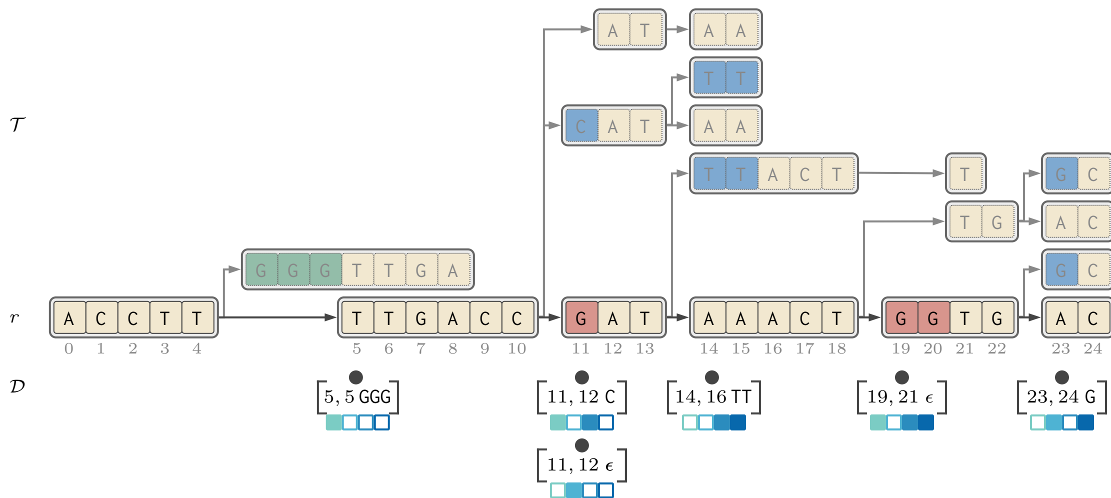


Figure 4.11: Sequence tree for the compact pangenome shown in Fig. 4.6 using a window size $\omega = 4$. The reference is represented by the labels of the nodes on the lowest level (black DNA symbols; surrounded with solid lines). The labels of nodes on higher levels are represented as journaled sequences (grey DNA symbols; surrounded with dotted lines), referencing either segments from the reference (yellow background) or alternate sequences from the sequence variants in $\mathcal{D}$.

As Fig. 4.11 already highlights, there are two types of nodes indicated by black and grey coloured symbols. This scheme is identical to the visualisation of a JS (see Section 4.3.4), meaning that segments of subtree nodes induced by a sequence difference of $\mathfrak{R}$ are not allocated explicitly but represented in a referentially compressed form using our journaling approach to space-efficiently view haplotype sequences. To acknowledge this, we refer to this virtual sequence tree as the *journaled sequence tree* (JST).

Fundamental terms and definitions

Expressed more formally, a JST is a node-labelled *directed acyclic graph* (DAG), denoted by $\mathcal{T} = (V, E)$, whose nodes are labelled with sequences representing either a segment from the reference sequence or an alternate sequence from one of the sequence differences in the underlying RCMS.

Following this observation, we divide the node set into a non-empty set of reference nodes, denoted by V_{ref}, and a possibly empty set of alternate nodes, denoted by V_{alt}, such that $V = V_{\text{ref}} \cup V_{\text{alt}}$, and where $V_{\text{ref}} \cap V_{\text{alt}} = \emptyset$. In the subsequent sections, we shall use this distinction to specify different node operations. Furthermore, the set V_{ref} always consists of at least one node which represents the root node of $\mathcal{T}$. We write $root(\mathcal{T}) \in V_{\text{ref}}$ to denote this special node. The size of a JST is defined as $|\mathcal{T}| = |V|$.

Two nodes v and w are connected via a directed edge $e = (v, w) \in V \times V$. We call the first node (i.e. v) the *head* and the second node (i.e. w) the *tail* of e, and we write $head(e) = v$ and $tail(e) = w$ to refer to the head and tail node, respectively.

We further differentiate between three types of edges:

1. $e \in V_{\text{ref}} \times V_{\text{ref}}$: the head and tail are both reference nodes.

2. $e \in V_{\text{ref}} \times V_{\text{alt}}$: the head is a reference node and the tail is an alternate node.

3. $e \in V_{\text{alt}} \times V_{\text{ref}}$: the head is an alternate node and the tail is a reference node.

In general, the in-degree and out-degree, i.e. the number of incoming and outgoing edges, of all inner reference nodes can be at most 2, while all inner alternate nodes always have an in-degree of 1 and an out-degree of at most 1. Alternate leaf nodes have an out-degree of 0. We use this constraint to reduce the complexity of the code, knowing that most sequence differences are embedded in two segments of the reference sequence, and therefore an alternate node is only reachable from a reference node and must lead back to a reference node.

A path from node u to w in $\mathcal{T}$ is an ordered sequence $\pi = (v_0, v_1, \ldots, v_{k-1})$, with $u = v_0$ and $w = v_{k-1}$, of distinct nodes in V where $(v_i, v_{i+1}) \in E$ for all $i \in [0..k-1)$. The size of π is $k = |\pi|$ and we call a path π a *rooted path* if and only if its first node is also the root of $\mathcal{T}$, i.e. $v_0 = root(\mathcal{T})$.

Furthermore, we introduce the terms *reference path* and *alternate path*. The former term signifies a rooted path π where all its nodes are members of the reference node set, such that $\forall v \in \pi, v \in V_{\text{ref}}$ (dark grey edges in Fig. 4.12). Conversely, we say a rooted path is an alternate path if and only if there exists at least one node that is also a member of the alternate node set, i.e. $\exists v \in \pi, v \in V_{\text{alt}}$ (light grey edges in Fig. 4.12). In the following, we use the symbols π_{ref} and π_{alt} to discriminate between a reference path and an alternate path, respectively. Importantly, the definition of π_{alt} does not exclude the fact that another $u \in \pi_{\text{alt}}$, with $u \neq v$, can be a member of V_{ref}.

In addition, we use the term *alternate subtree*, denoted by $\mathcal{T}_{\delta_i} \subset \mathcal{T}$, to describe any subset of nodes as well as their respective edges that form a subtree rooted in an alternate node that corresponds to the sequence difference indicated by $\delta_i \in \mathfrak{R}$. More specifically, if $\mathcal{T}_{\delta_i} = (V', E')$ denotes an alternate subtree, then there exists an edge $e \in E$ such that $head(e) \notin V'$ and $tail(e) \in V'$, and additionally $head(e) \in \pi_{\text{ref}}$ and $tail(e) = root(\mathcal{T}_{\delta_i})$. This also means that every rooted path of an alternate subtree is also an alternate path because $root(\mathcal{T}_{\delta_i}) \in V_{\text{alt}}$.

Node representation

In this section, we describe the primary abstract representation of a node in our model. At its core, a node is an ordered pair consisting of a left and a right *breakpoint boundary*. A breakpoint boundary is itself modelled as an ordered pair $(\delta, \beta) \in \mathfrak{R} \times \{\mathrm{l}, \mathrm{h}\}$, where the first element represents a handle to the currently visited sequence difference in the underlying RCMS $\mathfrak{R}$, and the second element, i.e. β, is a value from the binary alphabet $\{\mathrm{l}, \mathrm{h}\}$ characterising whether the breakpoint boundary corresponds to the low (i.e. $\beta = \mathrm{l}$) or the high (i.e. $\beta = \mathrm{h}$) breakend of the breakpoint associated with δ. Given a node v, we write $\mathit{left}(v) = (\delta_i, \beta_{\mathrm{L}})$ to denote its left breakpoint boundary and $\mathit{right}(v) = (\delta_j, \beta_{\mathrm{R}})$ to denote its right breakpoint boundary. Moreover, $v \in V$ if and only if $i < j$ or $i = j$ and $\beta_{\mathrm{L}} = \mathrm{l}$ and $\beta_{\mathrm{R}} = \mathrm{h}$.

Having this, we redefine the primary fields *low*, *high*, *seq*, *cov* already known from a covered sequence difference δ for a node v as follows:

$$\mathit{low}(v) = \begin{cases} \mathit{low}(\delta_i) & \text{if and only if } \beta_{\mathrm{L}} = \mathrm{l} \\ \mathit{high}(\delta_i) & \text{if and only if } \beta_{\mathrm{L}} = \mathrm{h} \end{cases} \tag{4.15}$$

$$\mathit{high}(v) = \begin{cases} \mathit{low}(\delta_j) & \text{if and only if } \beta_{\mathrm{R}} = \mathrm{l} \\ \mathit{high}(\delta_j) & \text{if and only if } \beta_{\mathrm{R}} = \mathrm{h} \end{cases} \tag{4.16}$$

$$\mathit{cov}(v) = \begin{cases} \mathit{cov}(\delta_i) & \text{if and only if } v \in \pi_{\mathrm{alt}} \\ H_X & \text{if and only if } v \in \pi_{\mathrm{ref}} \end{cases} \tag{4.17}$$

$$\mathit{seq}(v) = \begin{cases} \mathit{seq}(\delta_i) & \text{if and only if } v \in V_{\mathrm{alt}} \\ r[\mathit{low}(v)..\mathit{high}(v)) & \text{if and only if } v \in V_{\mathrm{ref}} \end{cases} \tag{4.18}$$

The low field of v represents either the low or the high breakend of its left breakpoint boundary. Similarly, the high field of v represents either the low or the high breakend of its right breakpoint boundary. A node v also has a coverage field, which has different interpretations depending on whether the corresponding node is inside an alternate path or not. If the node is inside an alternate path, the coverage field represents the coverage of the sequence difference marking the left breakpoint boundary. Otherwise, it represents the full haplotype coverage H_X including all haplotypes.

Lastly, the sequence of a node v represents either the alternate sequence of the sequence difference of its left breakpoint boundary if v is a member of an alternate path, or the corresponding infix of the reference sequence r specified by its low and high breakend values if v is a member of the reference node set. Note that this distinction is irrespective of whether v belongs to a reference path or an alternate path. Figure 4.12 depicts the conceptual graph object obtained for the RCMS from Fig. 4.6.

Basic traversal operations

The presented node representation is sufficient to implement the basic operations needed to traverse the JST in an ordered way. However, in order to provide a proper start and end condition for the traversal, we introduce two artificial sequence differences that signal the

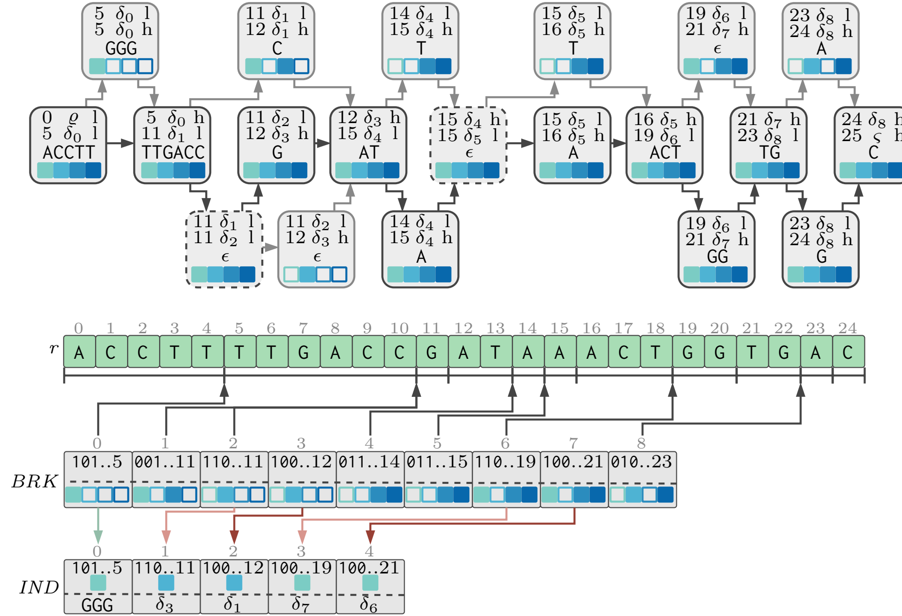


Figure 4.12: Conceptual graph generation for the RCMS given in Fig. 4.6. The breakends in *BRK* partition the reference into smaller segments. Each segment is described by a node in the graph. The nodes show the reference position and state of their low (first line) and high (second line) breakpoint boundary, the label sequence, and the coverage. Nodes with dark grey lines are members of V_{ref}, and with light grey lines are members of V_{alt}. The two reference nodes with dashed lines are inserted to provide a well-defined traversal over the graph.

low and high terminus of a given RCMS $\mathfrak{R}$. We use ϱ to mark the low terminal and ς for the high terminal. The fields of these artificial sequence differences are defined as follows:

$$
\begin{aligned}
pos(\varrho) &= 0 & pos(\varsigma) &= |r| \\
code(\varrho) &= \texttt{100} & code(\varsigma) &= \texttt{110} \\
low(\varrho) &= pos(\varrho) & low(\varsigma) &= pos(\varsigma) \\
high(\varrho) &= pos(\varrho) & high(\varsigma) &= pos(\varsigma) \\
seq(\varrho) &= \epsilon & seq(\varsigma) &= \epsilon \\
cov(\varrho) &= H_X & cov(\varsigma) &= H_X
\end{aligned} \tag{4.19}
$$

As can be seen, both terminals represent an empty alternate sequence mapping to a zero-sized breakpoint covered by all sequences of $\mathfrak{R}$. The low terminal points to position 0, and the high terminal to $|r|$. Moreover, their codes are chosen such that $\varrho < \delta < \varsigma$ for all $\delta \in \mathfrak{R}$. Consequently, we define the leftmost breakpoint boundary of any node in $\mathcal{T}$ as (ϱ, l), and the rightmost breakpoint boundary as (ς, h) (see Fig. 4.12).

Having this, we can now describe the primary operations to go from a node v to its successor w using the two algorithms `nextRef` and `nextAlt` listed in Algorithm 4.11 and Algorithm 4.12, respectively.

Next ref

Algorithm 4.11: `nextRef`

Input: Parent node v
Output: Child node w or `NIL`

1 $(\delta_j, \beta_\mathrm{R}) \leftarrow \mathit{right}(v)$
2 **if** $\delta_j = \varsigma$ **then**
3   **return** `NIL`
4 $k \leftarrow j + 1$
5 **while** $\delta_k \neq \varsigma$ **and** $\mathit{pos}(\delta_k) < \mathit{high}(v)$ **do**
6     $k \leftarrow k + 1$
7 $w \leftarrow v$
8 $\mathit{left}(w) \leftarrow (\delta_j, \beta_\mathrm{R})$
9 $\mathit{right}(w) \leftarrow (\delta_k, \mathit{bnd}(\delta_k))$
10 **return** w

As the name suggests, we use the **`nextRef`** operation to go to the next reference node w in V_ref from the current parent node v if such a successor node exists. At the beginning, the procedure reads the state of the right breakpoint boundary of the parent node v. If the associated sequence difference δ_j is equal to ς, the parent node v is a leaf and the procedure is terminated by returning `NIL`. Here, `NIL` refers to the special *null* state of nodes indicating that the parent v has no successor in V_ref. Otherwise, the algorithm goes to the upcoming element δ_k in $\mathfrak{R}$.

In case the position of the next element δ_k is less than the position of the right breakend boundary of v, the algorithm continues incrementing δ_k until it finds the first successor of δ_j whose low breakend is greater than or equal to the right position of the parent node (Lines 5 to 6). Notably, the while loop executes at most three times. That is, if the parent node $v \in V_\mathrm{alt}$, its high breakend might be larger than the position of the next sequence difference in the case that δ_j and δ_{j+1} point both to a SNV, because of our choice not to store the high breakends of these variant types. To compensate for this, we skip to the next sequence difference, whose position is greater than or equal to the high breakend of the right breakpoint boundary of the parent node.

The while loop is not executed for insertions or deletions, since in both cases, the condition $\mathit{pos}(\delta_{j+1}) < \mathit{high}(v)$ is always false. This holds for insertions, because their high breakend is equal to their low breakend and for deletions we use the breakend mate information (see Algorithm 4.12 in the subsequent section) to directly jump to the corresponding entry in $\mathfrak{R}$ that marks the end of the deletion. Therefore and because of Definition 4.3.5, we have that only SNVs at the same position must be skipped, which can be at most 3. If the parent node $v \in V_\mathrm{ref}$, then its right breakpoint boundary would represent the low breakpoint of δ_j and hence the position of the next SNV is equal to this position such that the while loop is skipped.

Having found a valid δ_k, the procedure makes a copy of v to create the child node w including all its internal states and resets its left breakpoint boundary to the right breakend boundary of its parent node. Thus, the left breakend boundary of w seemingly connects to the right

breakend boundary of v. Subsequently, the algorithm checks whether δ_k marks the end of the breakend dictionary, where

$$bnd(\delta) = \begin{cases} \mathrm{l} & \text{if and only if } pos(\delta) = low(\delta) \text{ or } \delta = \varrho, \\ \mathrm{h} & \text{if and only if } pos(\delta) = high(\delta) \text{ or } \delta = \varsigma. \end{cases} \quad (4.20)$$

Finally, the new child node w is returned.

Next alt

Algorithm 4.12: `nextAlt`

Input: Parent node v
Output: Child node w or `NIL`

```
1 (δj, βR) ← right(v)
2 if βR = h then // parent is non-branching node
3     return NIL
4 w ← v
5 left(w) ← (δj, βR)
6 right(w) ← (mate(δj), h)
7 return w
```

The second operation, `nextAlt`, is used to visit the child that corresponds to a particular sequence difference in $\mathfrak{R}$. In the beginning, this procedure checks if the right breakpoint boundary β_{R} represents the high breakend of the corresponding sequence difference δ_j. If this check evaluates to **true**, `NIL` is returned.

In the other case, the new child node w is created as a copy of v. Subsequently, its left breakpoint boundary is reset to the right breakpoint boundary of its parent v, and its right breakpoint boundary is set to the high breakend of its breakend mate. Notably, this method always produces nodes that are members of V_{alt}. As such, every alternate node in $\mathcal{T}$ represents the left and right breakend of a single sequence difference in $\mathfrak{R}$, i.e. $\delta_i = \delta_j$ and $\beta_{\mathrm{L}} = \mathrm{l}$ and $\beta_{\mathrm{R}} = \mathrm{h}$.

4.4.2 Tree polishing

Having described how we can traverse a given RCMS $\mathfrak{R}$ using tree-like traversal operations, we now focus on strategies to dynamically refine the size of the tree. These strategies aim to reduce the total number of nodes in the virtual JST and can be seen as additional adaptors of the essential operations `nextRef` and `nextAlt`. We refer to these strategies as *tree polishing operations*.

Among others, the three main tree polishing operations are: pruning of alternate subtrees based on the minimal required length of a *path sequence*, pruning of alternate subtrees based on the *path cover*, and maximal expansion of nodes with out-degree 1. In the following, we first describe the two approaches to prune the subtrees and subsequently discuss the node expansion method.

Minimal path sequence

Given a path $\pi = (v_0, v_1, \ldots, v_{k-1})$ in a JST $\mathcal{T}$, we define the *path sequence* as the sequence obtained by concatenating the sequences of all nodes in the path. Formally, we write:

$$seq(\pi) = seq(v_0) + seq(v_1) + \ldots + seq(v_{k-1}). \tag{4.21}$$

Let $\omega \in \mathbb{N}$, where $\omega \ll |r|$, be the size of a window used by an online algorithm α. When applying α on a given RCMS $\mathfrak{R}$ using a JST traversal, we want to make sure that the size of alternate subtrees are kept as small as possible while ensuring that no ω dependent sequence context around the sequence differences is missed. This can be achieved by bounding the length of alternate paths, starting at the root of an alternate subtree $\mathcal{T}_{\delta_i}$, based on the minimal required size of the associated path sequence. Specifically, we require, without loss of generality, that the minimal length of an alternate path π_{alt} passing through the root of an alternate subtree $\mathcal{T}_{\delta_i}$ is greater than or equal to $|seq(root(\mathcal{T}_{\delta_i}))| + \omega - 1$.

To address this, we have implemented a tree polishing routine called `cutAfter`, which augments a node in the JST with a journaled sequence representing the current visited path sequence, along with two offsets that project the left and right breakpoint boundaries to their corresponding positions within the journaled view. Additionally, there is a parameter tracking the remaining size of the path sequence. For a given node $v \in V$ in the JST, we use the fields $jnl(v)$, $\mathit{off}_{\text{L}}(v)$, $\mathit{off}_{\text{R}}(v)$, and $rem(v)$ to refer to these values.

The final algorithm, which augments the JST with the option to prune the tree after reaching a certain minimal length, is depicted in Algorithm 4.13.

Algorithm 4.13: `cutAfter`

Input: v, ω

1 **if** $v = $ `NIL` **or** $v \in \pi_{\text{ref}}$ **then**
2   **return** v
3 **if** $rem \leq 0$ **then** `// reached minimal path sequence size`
4   **return** `NIL`
5 **if** $v = root(\mathcal{T}_{\delta_i})$ **then** `// reset if root of alternate subtree`
6     $\mathit{off}_{\text{R}}(v) \leftarrow 0$
7     $rem(v) \leftarrow |seq(v)| + \omega - 1$
8 $rem(v) \leftarrow rem(v) - |seq(v)|$
9 $\mathit{off}_{\text{L}}(v) \leftarrow \mathit{off}_{\text{R}}(v)$
10 $\mathit{off}_{\text{R}}(v) \leftarrow \mathit{off}_{\text{R}}(v) + |seq(v)| - (high(v) - low(v))$
11 **if** $v \in V_{\text{alt}}$ **then**
12   `replace(`$jnl(v)$, $pos(v) + \mathit{off}_{\text{L}}(v)$, $seq(v)$`)`
13 **return** v

The `cutAfter` algorithm is invoked with a node and the minimal path sequence size ω. Then, `cutAfter` checks if v is `NIL` or a member of π_{ref}, as tree pruning is only performed on paths of alternate subtrees and not on the reference path. If either condition is met, the algorithm skips the polishing operation for v. Otherwise, it checks if the minimal path sequence was reached and returns `NIL` if this is true, or continues if not.

Next, the algorithm tests whether the current node v is the root of an alternate subtree $\mathcal{T}_{\delta_i}$. If this evaluates to **true**, $\mathit{off}_{\mathrm{R}}(v)$ is set to 0, and the remaining size of the path is initialised as the sum of the size of the current node sequence and the user-defined minimal path size $\omega - 1$.

Subsequently, the remaining size $\mathit{rem}(v)$ is updated by subtracting the size of the sequence associated with the current node v. Additionally, the last right offset value is assigned to the left offset value of v and the right offset value is updated by adding the effective size of the current segment represented by v. Following this, the corresponding position in the journaled view of the path sequence is computed using $\mathit{pos}(v) + \mathit{off}_{\mathrm{L}}(v)$. At this point, $\mathit{off}_{\mathrm{L}}(v)$ represents the cumulative sum of all effective sizes from sequence differences preceding the current node. Finally, the necessary modifications are applied to $\mathit{jnl}(v)$ to reflect the changes, but only if $v \in V_{\mathrm{alt}}$.

Unsupported paths

Another observation that can be made is that there may be paths within an alternate subtree whose path sequence does not exist in any of the encoded sequences in $\mathfrak{R}$. It would be beneficial to detect and skip such paths to optimise the tree traversal.

To address this, we introduce a path coverage and track this during the traversal. For a node $v \in V$ in $\mathcal{T}$, we augment it with an additional field $\kappa(v)$, which represents the haplotype coverage of the current path. The corresponding tree polishing algorithm, called `cutUncovered`, is depicted in Algorithm 4.14.

Algorithm 4.14: `cutUncovered`

Input: v

```
1  if v = NIL or v ∈ π_ref then
2  |  return v
3  if v ∈ V_alt then
4  |  κ(v) ← κ(v) ∩ cov(v)
5  else
6  |  κ(v) ← κ(v) \ cov(v)
7  if κ(v) = ∅ then
8  |  return NIL
9  else
10 |  return v
```

Similar to Algorithm 4.13, the algorithm `cutUncovered` proceeds only if the current node v is neither `NIL` nor from the reference path. It then checks if the new node is either an alternate node or a reference node. In the case of an alternate node, the algorithm computes the new path coverage as the set intersection between the current path coverage and the coverage of the sequence difference associated with the current node. On the other hand, if the new node is a reference node, the algorithm updates the path coverage by subtracting the haplotype coverage associated with the left breakpoint boundary of v (see Eq. (4.18)). This ensures that the path coverage remains consistent with the encoded haplotypes.

After updating the current path coverage, the algorithm checks if the new coverage is empty or not. In the former case `NIL` is returned effectively pruning unsupported haplotype paths, and in the latter case the updated node v is returned.

Node expansion

The last polishing strategy is based on the observation that nodes with out-degree 1, which includes all alternate nodes and non-branching reference nodes, can be expanded by merging them with their successor nodes to form a single node.

Algorithm 4.15: `expand`

Input: v

```
1 if v = NIL or β_R = l then
2 |   return v
3 v′ ← nextRef(v)
4 (δ_j, β_R) ← right(v′)
5 while v′ ≠ NIL and β_R = h do
6 |   v′ ← nextRef(v)
7 |   (δ_j, β_R) ← right(v′)
8 right(v) ← (δ_j, β_R)
9 return v
```

Algorithm 4.15 presents the corresponding implementation of the expansion strategy. The algorithm first checks if the current node is a branching node, which means its right breakpoint boundary refers to the low breakend of a sequence difference. In this case the node can not be expanded and is returned unmodified.

In all other cases, i.e. v is an alternate node or a reference node that connects to the high breakend of a deletion. In this case, the algorithm calls `nextRef` to obtain the next successor node v' and repeats this step until it finds a branching node or reaches the end of the current path. Once a branching node or the end of the path is reached, the algorithm replaces the right breakpoint boundary of v with the new breakpoint boundary represented by v', effectively coalescing all successor nodes with out-degree 1 into v.

4.4.3 Universal traversal

We can utilise the described traversal operations and tree polishing routines to implement a universal traversal scheme for the RCMS that can be used with any online algorithm processing a given sequence within an algorithm-specific window, independent of the window size ω. This scheme follows an *alt-first* pre-order traversal, where the alternate subtree is visited first before continuing with the corresponding reference part whenever an inner node with out-degree two is encountered.

During each iteration, the algorithm is invoked with a segment of the current path sequence whose size depends on the window size and specific features of the given algorithm. We distinguish between two traversal variants: a *state-oblivious traversal* and a *resumable traversal*.

State oblivious traversal

Algorithm 4.16: State oblivious JST traversal

Input: $\mathfrak{R}$, α

```
 1  makeRoot():
 2      v_root ← ((ϱ,1), (δ_0,1))
 3      jnl(v_root) ← (0, r)
 4      off_L(v_root) ← off_R(v_root) ← 0
 5      κ(v_root) ← cov(ϱ)
 6      return v_root
 7  getSlice(v, ω):
 8      x ← view(jnl(v))
 9      i ← max(0, low(v) + off_L(v) − (ω − 1))
10      j ← high(v) + off_R(v) + min(rem(v), 0)
11      return x[i..j]
12  ω ← |α|
13  𝒯 ← ∅
14  push(𝒯, makeRoot())
15  while π ≠ ∅ do
16      v ← pop(𝒯)
17      α(getSlice(v, ω))
18      w ← cutUncovered(cutAfter(expand(nextRef(v)), ω))
19      if w ≠ NIL then
20          push(𝒯, w)  // w ∈ V_ref, (v, w) ∈ E
21      w′ ← cutUncovered(cutAfter(expand(nextAlt(v)), ω))
22      if w′ ≠ NIL then
23          push(π, w′)  // w′ ∈ V_alt, (v, w′) ∈ E
```

The state-oblivious traversal variant is suitable when the online algorithm is treated as a black box, meaning that its internal state is hidden in the implementation. This is often the case when using algorithms from third-party libraries where exposing the state was not part of the original API. In this scenario, we introduce the possibility of including a user-defined path prefix immediately preceding the currently visited node. In case of online pattern matching, this prefix includes the $\omega - 1$ contiguous symbols immediately preceding the left breakpoint boundary of the last node of the root path. The extended label of a visited node represents an independent search sequence for the applied online algorithm. The details of this process are shown in Algorithm 4.16

We invoke the state-oblivious traversal with an instance of $\mathfrak{R}$ and an online algorithm α. At the beginning of the algorithm (starting at Line 12), we obtain the window size of α, denoted by $|\alpha|$. We require that $|\alpha| \in \mathbb{N}$; otherwise, we can assume that the algorithm is not properly initialised. Next, we initialise a stack $\mathcal{T}$ that tracks the currently visited path of the conceptual sequence tree. We push the root node onto the stack to start the traversal.

The root node is configured in the `makeRoot` routine (Lines 1 to 6). In this routine, a new node v_{root} is created, representing the root of the JST. Its left breakpoint boundary represents the left terminal ϱ, and its right breakpoint boundary represents the low breakend of the first sequence difference in $\mathfrak{R}$. Note that according to the invariants of the breakend

dictionary, there can not be a high breakend before the first low breakend in BRK. The journal of v_{root} is initialised with a single entry representing the whole reference sequence r, and the corresponding left and right offsets are set to 0. Lastly, the path coverage $\kappa(v_{\text{root}})$ is set to the coverage of its left breakpoint boundary, which, according to Eq. (4.19), is the full coverage. The initialised root node is then returned.

After initialising the stack $\mathcal{T}$, the algorithm performs the following steps repeatedly until $\mathcal{T}$ is empty:

1. Pop the current node v from $\mathcal{T}$.
2. Invoke the algorithm α on the respective segment determined by `getSlice`(v, ω).
3. Visit the reference child node of v by calling `nextRef`(v) and apply the tree polishing procedures on the returned child node w.
4. Push the polished child node w to $\mathcal{T}$ if it is not `NIL`.
5. Visit the alternate child node of v by calling `nextAlt`(v) and apply the tree polishing procedures on the returned child node w'.
6. Push the polished child node w' to $\mathcal{T}$ if it is not `NIL`.

The `getSlice` routine is shown in Lines 7 to 11 of Algorithm 4.16. In the oblivious case, we obtain the view of the journaled path sequence associated with the passed node v. This always represents the path sequence of the root path because, as a journaled sequence, we maintain the prefix of r before the left breakpoint boundary of any alternate subtree root.

Correspondingly, we compute the start position of the segment of this path sequence, denoted by i, as the maximum of 0 and the current low breakend $low(v)$ plus the stored left offset $\mathit{off}_{\mathrm{L}}(v)$ to account for the recorded insertions and deletions. We subtract from this the given length ω, which is $\omega - 1$ in this case. The corresponding end position of the segment, denoted by j, is computed as the high breakend $high(v)$ plus the right offset $\mathit{off}_{\mathrm{R}}(v)$ plus the minimum of $rem(v)$ and 0.

According to our implementation of the `cutAfter` polishing routine, $rem(v)$ can only become less than or equal to 0 if v is a leaf node of an alternate path. In this case, it may happen that the right breakpoint boundary of v is beyond the minimum position needed for the current window size. To correct this, we adjust it with $rem(v)$, which corresponds to the negative distance between the minimum position and the position indicated by the right breakpoint boundary. Finally, the corresponding segment $x[i..j)$ over the root path sequence is returned.

A call to the polished `nextRef` can only return a new reference node or `NIL`. If it is not `NIL`, it is added to the stack $\mathcal{T}$ before the corresponding alternate node is visited and added to $\mathcal{T}$, unless it is `NIL` as well. This ensures that if v has a successor $w' \in V_{\text{alt}}$, then w' is processed before w in the next iteration. Therefore, the nodes of the reference path π_{ref} are always processed after the respective alternate subtree has been traversed. As a result, we guarantee that the last node visited is a reference node in π_{ref}, and its right breakpoint boundary is (ς, h).

Resumable traversal

Algorithm 4.17: Resumable JST traversal

Input: $\mathfrak{R}$, α

```
1  ω ← |α|
2  T ← ∅
3  A ← ∅
4  push(T, makeRoot())
5  push(A, capture(α))
6  while π ≠ ∅ do
7      v ← pop(T)
8      restore(α, pop(A))
9      α(getSlice(v, 0))
10     w ← cutUncovered(cutAfter(expand(nextRef(v)), ω))
11     if w ≠ NIL then
12         push(T, w)  //  w ∈ V_ref ,  (v, w) ∈ E
13         push(A, capture(α))
14     w′ ← cutUncovered(cutAfter(expand(nextAlt(v)), ω))
15     if w′ ≠ NIL then
16         push(π, w′)  //  w′ ∈ V_alt ,  (v, w′) ∈ E
17         push(A, capture(α))
```

In the special case where we can capture and restore the inner state of the given algorithm α, we can refine the traversal routine to narrow the segment over the path sequence. Specifically, we no longer need to include the ω symbols before the left breakpoint boundary of the currently visited node v in the `getSlice` subroutine.

To access this traversal version, we require the existence of two functions, `capture` and `restore`, such that the expression $\alpha = \texttt{restore}(\alpha, \texttt{capture}(\alpha))$ preserves equality. In other words, we can obtain a copy of α and its state, and return to this state at a later point in time by restoring α from it. Note that this feature is implicitly given if α is copyable. However, it is often more efficient to provide a specialisation of these functions to limit the size of the state.

With these requirements fulfilled, we introduce a second stack, denoted as A, which stores the captured states of α every time a new child node is added to $\mathcal{T}$. Similar to $\mathcal{T}$, A is initialised with the initial state of α. Before applying the algorithm to the next path sequence segment, we restore α from its last captured state stored in A. Since α now represents the exact state before the current node v is processed, we can call `getSlice` in such a way that only the sequence of v between its left and right breakpoint boundaries is passed to α. The rest of the traversal remains the same. The corresponding pseudocode is listed in Algorithm 4.17.

4.4.4 Tree transformation and tracing

In this section, we discuss additional transformations and features of the JST that we implemented to increase its range of applications. These features include:

1. The ability to apply the mentioned traversals in the backward direction by constructing a reversed JST.
2. The ability to split a JST into a set of subtrees.
3. Providing a mechanism to efficiently traceback to a visited alternate path at a later point in time.

Reversed JST

Notably, we can reuse the previously described algorithms to traverse the JST in the backward direction, i.e. from right to left, by simply providing a lightweight wrapper that offers a reversed view of the underlying RCMS object. We denote the reversed counterpart of $\mathfrak{R}$ as $\reflectbox{$\mathfrak{R}$}$.

To obtain this representation, it suffices to reverse the reference sequence as well as the alternate sequences, and the access to the stored sequence differences while adjusting their fields accordingly.

Definition 4.4.1 (Reverse sequence)**.** Let $x \in \Sigma$ be a sequence of length $n = |x|$. The reversed representation of x, denoted as $\texttt{reverse}(x)$, is obtained as follows: $\texttt{reverse}(x) = x_{n-1}x_{n-2}\dots x_0$.

With this definition, we define a mapping from the i-th element in $\reflectbox{$\mathfrak{R}$}$, denoted as $\reflectbox{$\delta$}_i$, to the corresponding element in the wrapped forward-oriented $\mathfrak{R}$ such that $\reflectbox{$\delta$}_i = \delta_{n_\mathrm{B}-1-i}$, where $n_\mathrm{B} = |BRK|$ denotes the number of breakends stored in the breakend dictionary of $\mathfrak{R}$. We redefine the fields *pos*, *low*, *high*, and *alt* of $\reflectbox{$\delta$}_i$ as follows:

$$
\begin{aligned}
pos(\reflectbox{δ}_i) &= |r| - pos(\delta_{N-1-i}) \\
low(\reflectbox{δ}_i) &= |r| - high(\delta_{N-1-i}) \\
high(\reflectbox{δ}_i) &= |r| - low(\delta_{N-1-i}) \\
alt(\reflectbox{δ}_i) &= \texttt{reverse}(alt(\delta_{N-1-i})).
\end{aligned}
\tag{4.22}
$$

Additionally, we reverse the reference sequence, i.e. $\reflectbox{$r$} = \texttt{reverse}(r)$, so that the mapped breakends of $\reflectbox{$\delta$}_i$ now correspond to the correct positions within $\reflectbox{$r$}$. Specifically, the high breakend of the original sequence difference corresponds to the low breakend in the reversed RCMS, and vice versa. The resulting reversed representation of a JST, denoted as $\reflectbox{$\mathcal{T}$}$, is illustrated in Fig. 4.13.

It is worth noting that this modification did not impact the coverage of the traversal approaches, which remains the same. Additionally, none of the original data structures needed to be copied, as the traversal algorithms can be applied directly to the reversed view of the RCMS.

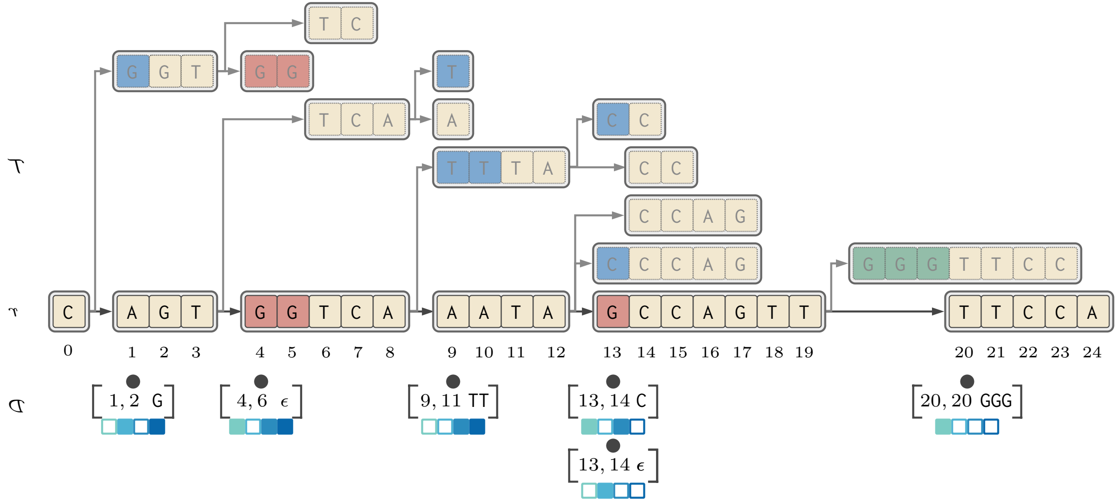


Figure 4.13: Reversed JST $\daleth$ obtained from reversing the example RCMS $\mathfrak{R}$ shown Fig. 4.11.

Partial JST

The second tree transformation that we implemented is the partitioning of an RCMS $\mathfrak{R}$ into multiple chunks, which can then be processed independently in so called *partial journaled sequence trees*. More specifically, this transformation involves extracting a subset of nodes and edges from the original JST that correspond to a segment of the reference sequence. To define this segment, we specify the start position in the reference sequence as $p \in [0..|r|)$ and the length of the reference path sequence following this position as $l \in [0..|r| - p)$. The extracted subset of nodes and edges, denoted as $\mathcal{T}_{p,l} \subseteq \mathcal{T}$, forms the *partial journaled sequence tree* (PJST) corresponding to the segment $r_{p..p+l}$.

Similar to the reversed transformation, we implemented this as a lightweight wrapper. Specifically, we modified the creation of the root and sink nodes of the PJST $\mathcal{T}_{p,l}$ in the way that we adapted the positions of the leftmost and rightmost breakpoint boundaries (see Eq. (4.19)) of $\mathcal{T}_{p,l}$ to:

$$pos(\varrho_{p,l}) = p \tag{4.23}$$
$$pos(\varsigma_{p,l}) = p + l \tag{4.24}$$

With these modifications, we redefined the `makeRoot` subroutine as shown in Algorithm 4.18.

Algorithm 4.18: `makeRoot` for partial JST

Input: $\mathfrak{R}$, p, l

1 $\delta_i \leftarrow$ lower bound such that $pos(\delta_{i-1}) < p \leq pos(\delta_i)$
2 $v_{\text{root}} \leftarrow ((\varrho_{p,l}, 1), (\delta_i, bnd(\delta_i)))$
3 $jnl(v_{\text{root}}) \leftarrow (0, r)$
4 $\mathit{off}_{\text{L}}(v_{\text{root}}) \leftarrow \mathit{off}_{\text{R}}(v_{\text{root}}) \leftarrow p$
5 $\kappa(v_{\text{root}}) \leftarrow cov(\varrho_{p,l})$
6 **return** v_{root}

First, we search for the lower bound δ_i in $\mathfrak{R}$, which takes logarithmic time, i.e. $\mathcal{O}(\log|BRK|)$. Once δ_i is determined, we set the left breakpoint boundary of the root node v_{root} to represent the low breakend of the artificial terminal $\varrho_{p,l}$, and the right breakpoint boundary to the corresponding breakend of the first sequence difference whose position is greater than or equal to p, i.e. $p \leq pos(\delta_i)$.

The associated journal to represent the path sequence is still initialised the same way, using the full reference sequence. However, we set both off_{L} and off_{R} to p in order to retrieve only the segment of the root node that starts at position off_{L} in the journaled view. Additionally, the path cover κ is initialised to represent all sequences. Due to this modified initialisation of the root node, we can reuse the existing traversal implementations as we fast-forward into the JST to the corresponding node in the reference path π_{ref} that spans the position p.

Next, we need to define the sink node of $\mathcal{T}_{p,l}$ in order to provide a proper condition to terminate the traversal at the correct position defined by $\varsigma_{p,l}$. However, this end condition should only be tested while traversing the reference path π_{ref}. Specifically, we ignore the artificial rightmost breakpoint boundary whenever we are in an alternate path that started before the position of $\varsigma_{p,l}$. Since we have access to the wrapped $\mathfrak{R}$, we can traverse all alternate subtrees whose root nodes start inside the given reference interval, even if an alternate path exceeds the artificial end position (e.g. due to a long deletion spanning the end position).

In addition, we also need to ensure that when an algorithm is applied to a $\mathcal{T}_{p,l}$, the $\omega - 1$ symbols following the rightmost breakpoint boundary defined by $\varsigma_{p,l}$ are included, unless $\varsigma_{p,l}$ is the global end terminal, i.e. $pos(\varsigma_{p,l}) = pos(\varsigma)$. This means that $\varsigma_{p,l}$ reflects the rightmost end position for the head of the window on which an algorithm is operating. To correctly model this, we wrapped the `nextRef` procedure as listed in Algorithm 4.19.

Algorithm 4.19: Wrapped `nextRef` for PJST

Input: v

```
1  if δ_R = ς_{p,l} then
2  |   left(v) ← right(v)
3  |   right(v) ← nxt(v)
4  |   if v ∈ π_alt then
5  |   |   w ← v
6  |   |   return w
7  w ← nextRef(v)  // call nextRef of base
8  if w ≠ NIL and w ∈ π_ref then
9  |   if low(w) ≤ pos(ς_{p,l}) < high(w) then
10 |   |   nxt(w) ← right(w)
11 |   |   right(w) ← (ς_{p,l}, 1)
12 |   else if pos(ς_{p,l}) < low(w) then
13 |   |   return NIL
14 return w
```

As mentioned above, the `nextRef` routine of the PJST wraps the `nextRef` routine (see Algorithm 4.11) of the basic implementation. Correspondingly, it delegates the call to its base implementation in Line 7 of Algorithm 4.19 to obtain the new child node w.

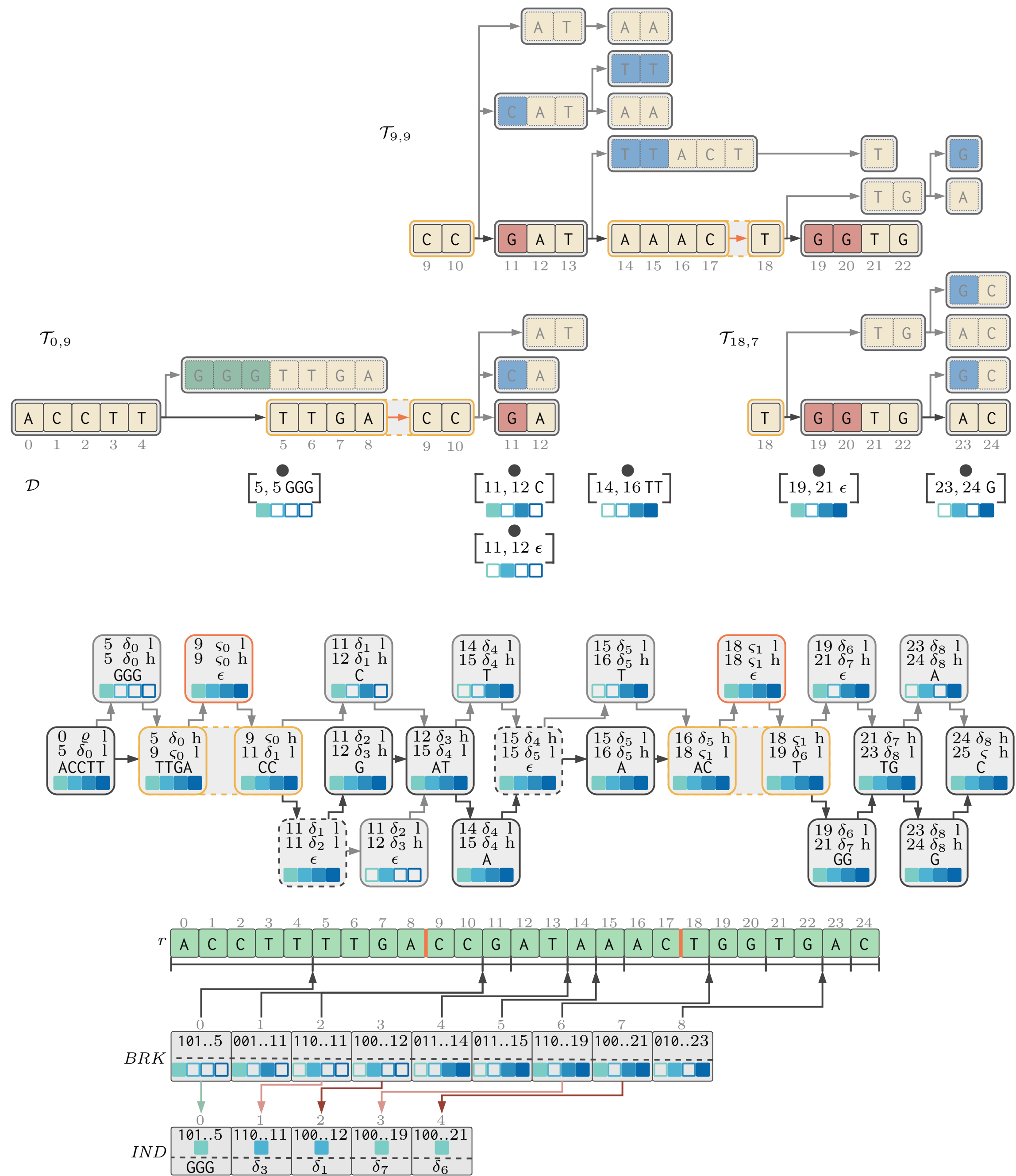


Figure 4.14: Chunked RCMS $\mathfrak{R}$ using a chunk size of 9, s.t. r is split at position 9 and 18 (orange lines). Orange nodes represent the artificially inserted terminal differences ς_0 and ς_1. Yellow lines mark the original nodes that are split due to the inserted terminals. For each chunk a partial JST can be constructed and traversed independently, while the artificial terminals induce alternate subtrees that are expanded just like regular ones. Here this is demonstrated with $\mathcal{T}_{0,9}$, $\mathcal{T}_{9,9}$, and $\mathcal{T}_{18,7}$, which all use a window size of 4.

Subsequently, it checks if w represents the last node in the reference path of $\mathcal{T}_{p,l}$ or if it is the sink node already. The former case is **true** if the reference segment represented by w overlaps the end position of $\mathcal{T}_{p,l}$ (Line 9). In this case, the original right breakpoint boundary of w is cached in its auxiliary field *nxt* and then overwritten with the artificial right breakpoint boundary $(\varsigma_{p,l}, \mathrm{l})$. We select the l state to indicate that w is a branching node. In other words, we treat $\varsigma_{p,l}$ as a real sequence difference, allowing the traversal algorithm to span an alternate subtree with $\varsigma_{p,l}$ as its root when calling the subsequent `nextAlt` routine. By the way we initialised $\varsigma_{p,l}$, we guarantee that adding this additional alternate subtree will not change any of the original sequence content encoded in the RCMS. However, we can now utilise the `cutAfter` polishing routine to expand $\mathcal{T}_{p,l}$ by exactly ω symbols that follow the empty alternate sequence at position $high(\varsigma_{p,l})$ at its end, including all possible alternate paths if there are other sequence differences in this particular window. See Fig. 4.14 for an example.

If w is not the last reference node, it is checked whether it represents the sink node, which is the case if the low breakend of the next node in π_{ref} is strictly greater than the final position $pos(\varsigma_{p,l})$. If this condition is **true**, `NIL` is returned. Otherwise, if w is some node in π_{ref} before the last reference node of $\mathcal{T}_{p,l}$ or if w is either `NIL` or an element of an alternate path, the algorithm proceeds without modifying w.

To account for the special right breakpoint boundary added in the case that w was the last node of the reference path, an additional correction step is added before the call to the base `nextRef` routine. If the right breakpoint boundary of the current parent node v is equal to the artificially set terminal boundary $(\varsigma_{p,l}, \mathrm{l})$, its left breakpoint boundary is set to the terminal boundary, and its right breakpoint boundary is restored from the cached original right breakpoint boundary $nxt(v)$. If v lies on an alternate path, we know that v represents the alternate subtree root spawned at the artificial terminal sequence difference $\varsigma_{p,l}$. In this case, we correct our intervention made earlier in the traversal by copying v with its restored right breakpoint boundary and return this copy as the new reference child node w. If v is not on an alternate path, the corrected v must have been a node of the reference path.

After calling the base `nextRef` routine, the returned child node w has its left breakpoint boundary set to the former right breakpoint boundary, whose position was strictly greater than $pos(\varsigma_{p,l})$. As a result, the `nextRef` routine of the PJST will return `NIL`, thereby terminating the traversal.

Tracing

The last feature we implemented for the JST traversal in this thesis is the ability to traceback to any traversed path using an auxiliary *seeking index*. This allows us to employ indexing to swiftly reconstruct the corresponding path of the JST captured during the traversal, similar to storing the position of a plain sequence.

We added this tracing feature by implementing another thin wrapper class around the JST interfaces. This wrapper augments a node with two additional fields: $idx \in [0..|BRK|)$, representing the index of the breakend in the breakend dictionary of the underlying RCMS $\mathfrak{R}$, and *dsc*, representing a descriptor to reconstruct the exact path starting at the alternate root defined by δ_{idx}.

During a traversal over a JST $\mathcal{T}$, we capture the current index of the currently visited sequence difference δ_i if δ_i marks the root of the alternate subtree spawned at that position, i.e. $idx(v) = i$, or if δ_i is the sequence difference of the right breakpoint boundary of a reference path node.

To differentiate between an alternate path and a node of the reference path, we use the value of the path descriptor *dsc*, which is implemented as a dynamically growing bitset. Initially, the path descriptor is empty. When a new alternate subtree root is created, we initialise the path descriptor of the current node to 1; otherwise, we keep it empty. After moving to the next child node within an alternate subtree, we append a 1 to the path descriptor if the new child node is an element of V_{alt}, and append a 0 otherwise. The length of the *dsc* determines the length of the alternate path starting at the alternate subtree root.

To reconstruct a path from a stored seeking index (idx, dsc), we use *idx* to create a new reference node w whose right breakpoint boundary represents the low breakend of δ_{idx}. If the path descriptor is empty, we know that the index came from node w of the reference path, and we are done. Otherwise, we need to unwind the alternate path encoded by *dsc*. We achieve this by iterating over the positions of *dsc*, and whenever we read a 1, we call `nextAlt` on the currently visited node; for a 0, we call `nextRef`. Since the first value of a non-empty *dsc* must always be 1, we initiate the alternate path by calling $\texttt{nextAlt}((w))$.

For example, suppose we are currently traversing the alternate subtree induced by δ_6 (deletion: $[19, 20, \epsilon]$) at position 19 in $\mathcal{T}$. Right after entering the alternate subtree root u, we have $idx(u) = 6$ and $dscu = \texttt{1}$. Now, consider the case where we visit the alternate path at δ_8, i.e. $v = \texttt{nextAlt}(u)$, and the corresponding reference path, i.e. $w = \texttt{nextAlt}(u)$. We track these paths such that $idx(v) = idx(w) = 6$, $dsc(v) = \texttt{11}$, and $dsc(w) = \texttt{10}$.

4.5 Application and evaluation

In this section, we demonstrate the versatility and applicability of our methods designed around the RCMS data structure by discussing various use cases that are relevant in the pangenomic area. To do so, we implemented a set of utility tools called *JuST tools* (Journaled Sequence Tree tools). These tools include the creation of an RCMS object from a given VCF file and the corresponding reference sequence file (`just::create`), simulation of reads (`just::sim`), extracting individual haplotypes from a given RCMS object (`just::view`), creating an IBF for a given RCMS object (`just::index`), and a prototypic read mapper (`just::map`).

Additionally, we created a benchmark suite to evaluate various configurations of the implemented RCMS routines. In the subsequent section, we will describe the data used for the benchmarks. Following that, we will present the results for pure online pattern matching based on the JST traversal, and then evaluate the prototypic read mapping approach.

4.5.1 Benchmark data

We used data from the 1KGP to analyse the performance of our methods. Specifically, we obtained the variant data from phase 3 of the 1KGP, which was generated by remapping and recalling the reads on the reference genome build GRCh38. The variant data can be accessed at the following FTP link: `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data_`

collections/1000_genomes_project/release/20190312_biallelic_SNV_and_INDEL/. We also used the GRCh38 reference genome build, which can be downloaded from: http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/reference/GRCh38_reference_genome/GRCh38_full_analysis_set_plus_decoy_hla.fa [Lowy-Gallego et al., 2019; Zheng-Bradley et al., 2017].

The dataset from the 1KGP comprised a total of 2,548 samples from 26 populations, resulting in a total of 5,096 haplotypes. We downloaded the VCF files for each individual chromosome and transformed them into an RCMS object for further analysis.

	VCF disk [GB]	RCMS disk [GB]	RCMS memory [GB]	time [s]	# SNVs	# InDels
chr1	60 (0.992)	4.1 (0.759)	4.487	183	5 795 045	396 787
chr2	66 (1.001)	4.4 (0.737)	4.76	203	6 356 815	433 735
chr3	55 (0.910)	3.7 (0.676)	4.062	175	5 280 535	360 957
chr4	53 (0.915)	3.9 (0.695)	4.259	167	5 120 663	357 146
chr5	50 (0.821)	3.3 (0.588)	3.551	154	4 788 373	326 662
chr6	47 (0.824)	3.4 (0.607)	3.704	149	4 539 753	323 583
chr8	43 (0.706)	2.9 (0.513)	3.296	133	4 162 376	263 072
chr7	44 (0.744)	3.0 (0.552)	3.095	136	4 222 930	288 477
chr9	33 (0.549)	2.7 (0.420)	2.917	102	3 642 067	245 147
chr10	38 (0.642)	2.7 (0.49)	2.911	115	3 632 296	241 962
chr11	38 (0.628)	2.5 (0.492)	2.453	117	3 500 317	239 723
chr12	36 (0.611)	2.3 (0.470)	2.742	112	3 178 998	205 361
chr13	27 (0.462)	2.1 (0.375)	2.22	83	2 575 087	185 757
chr14	25 (0.415)	1.8 (0.317)	1.874	77	2 383 124	165 778
chr15	23 (0.372)	1.6 (0.295)	1.696	69	2 153 931	147 521
chr16	25 (0.399)	1.7 (0.304)	1.769	76	2 410 530	142 465
chr17	22 (0.351)	1.6 (0.288)	1.532	66	2 066 683	142 176
chr18	22 (0.361)	1.4 (0.261)	1.668	67	2 047 352	138 389
chr19	18 (0.294)	1.2 (0.215)	1.263	54	1 625 697	113 126
chr20	17 (0.294)	1.2 (0.206)	1.269	52	1 706 441	111 050
chr21	11 (0.180)	0.772 (0.137)	0.779	31	976 598	68 670
chr22	11 (0.177)	0.723 (0.132)	0.828	31	993 879	65 199
total	764 (12.648)	52.995 (9.529)	57.135	2352	73 159 490	4 962 743

Table 4.3: Summary of transforming VCF files from 1KGP into RCMS objects per chromosome. Numbers in parenthesis represent gzipped file sizes.

Table 4.3 summarises general statistics about processing the downloaded VCF data per chromosome. The *input size* column represents the disk space occupied by the original VCF files. The following two columns indicate the disk space and memory capacity required for the corresponding RCMS objects. The *time* column depicts the time taken to transform the VCF input into an RCMS object using a single thread. The last two columns represent the number of SNVs and InDels contained in the final RCMS objects. Overall, InDels account for approximately 6.8% of all variants.

Loading the largest chromosome 2 into RAM using the current version of the RCMS implementation requires only 4.76 GB of main memory. The entire human genome data, including all chromosomes, could be loaded with less than 60 GB of RAM. This means that we need approximately 0.00136 bytes per input variant.

Simulated data sets

To evaluate the online pattern matching algorithms, we used `just::sim` to simulate four sequences (DSD32, DS64, DS128, and DS256) of lengths 32, 64, 128, and 256, respectively, from chr22. For the read mapping procedure evaluation, we simulated a read data set (DS100x100Ke3) for chr22 consisting of 100 000 reads of length 100 with an error rate of up to three errors (3% error rate).

To simulate the reads, we implemented a JST wrapper class that computed various statistics about the JST, such as the total number of traversed bases, the number of alternate subtrees, and the average subtree depth. We used this information to generate a list of candidate positions from which to sample the reads. In the second run, we traversed the JST again and sampled the reads from the generated positions.

Test system

We conducted all experiments on a 2-socket system equipped with Intel® Xeon® Platinum 9242 Processors. This system provided in total 96 physical cores and 192 logical cores due to Intel's hyper threading feature. The binaries were compiled in release mode using g++-12.1.0.

4.5.2 Online pattern matching

The first primary use case that we evaluated with our methods was the application of online pattern matching. As mentioned earlier, one of our main goals was to provide a generalised approach to execute any kind of online algorithm without the need to implement a specialised version that operates on an RCMS object. For this purpose, we used the pattern matching algorithms implemented in SeqAn2, namely `seqan::Horspool`, `seqan::ShiftOr`, and `seqan::Myers`. We wrapped these classes in a thin adapter class to provide the correct calling interface for the state-oblivious JST traversal. Additionally, we added a thin adapter class for the ShiftOr and Myers patterns to also provide a state-resumable version. We then searched each of the sampled sequences (DS32, DS64, DS128, and DS256) using the respective JST traversal approach for each of the pattern matching algorithms and compared it with the plain version. In the plain version, we iteratively applied the pattern matching algorithms to each haplotype encoded in the RCMS object using its journaled sequence representation. The final results are listed in Table 4.4.

It can be observed that the runtime of the algorithms using the JST traversal depends mostly on the lengths of the searched sequence. The longer the sequence to search, the longer the expected runtime. This is expected because longer sequences require traversing longer alternate paths and executing more tree polishing operations, resulting in a higher overall workload.

It is also noteworthy that the workload for the pattern matching algorithm increases as well. Specifically, for the ShiftOr and Myers algorithms, if the size of the pattern sequence exceeds the word size (i.e. longer than 64 bases), the workload increases with each additional word boundary. Looking at the plain execution modes, we see that the runtime of the ShiftOr algorithm for DS128 doubles, and for DS256, the runtime increases by a factor of 4

		DS32		DS64		DS128		DS256	
		time [s]	speedup	time [s]	speedup	time [s]	speedup	time [s]	speedup
Horspool	plain	867.557	1	514.948	1	783.08	1	585.25	1
	oblivious	7.696	112.72	11.364	45.31	19.441	33.8	36.432	16.06
ShiftOr	plain	742.13	1	739.52	1	1510.54	1	2278.35	1
	oblivious	7.546	98.34	11.636	63.55	23.709	63.71	63.191	36.05
	resumable	7.093	104.63	10.748	68.81	19.103	79.07	36.949	61.66
Myers	plain	2313.01	1	2166.36	1	5053.28	1	5074	1
	oblivious	9.438	245.07	14.488	149.53	35.376	142.84	89.108	56.94
	resumable	8.185	282.59	12.14	178.44	22.742	222.2	42.272	120.03

Table 4.4: Run time evaluation of different online pattern matching algorithms using JST traversal (oblivious and resumable rows) on the RCMS representation of chr22. The plain row depicts the base run time obtained from executing the respective pattern matching algorithm on all 5096 journaled sequences.

compared to DS32 and DS64. For the Myers algorithm, the runtime also doubles when the sequence length reaches 128 bases. However, for DS256, the runtime does not increase any further. This can be explained by the additional Ukkonnen trick, which allows skipping remaining cells of the currently computed DP matrix column if an optimal alignment within the given score limit cannot be reached for the currently aligned needle prefix.

In contrast, the workload of Horspool strongly depends on the distribution of symbols in the pattern sequence. As a result, we observed different run times in the plain execution for the different sample sequences. Nonetheless, the differences in run time between the Horspool and resumable ShiftOr pattern matching algorithms are negligible in our experiments. Moreover, for longer pattern sequences, the differences between the oblivious and resumable traversal strategies become more significant. For DS256, the resumable Myers and ShiftOr algorithms take roughly half the time compared to their oblivious counterparts. These results are highlighted in Figure 4.15.

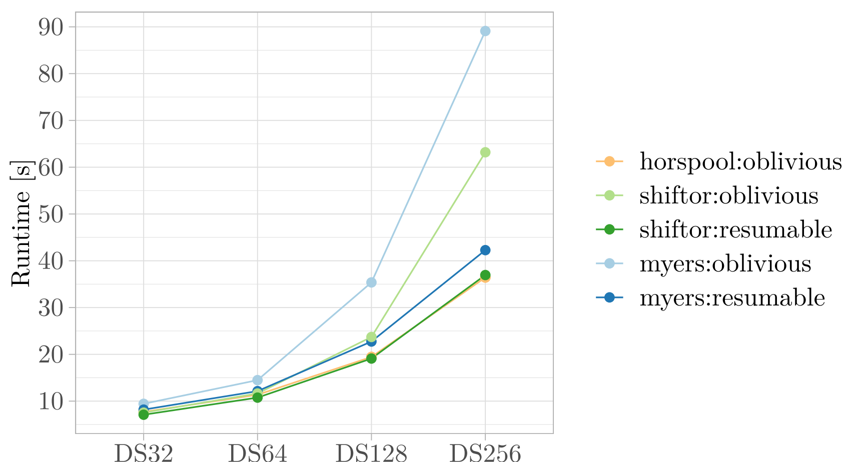
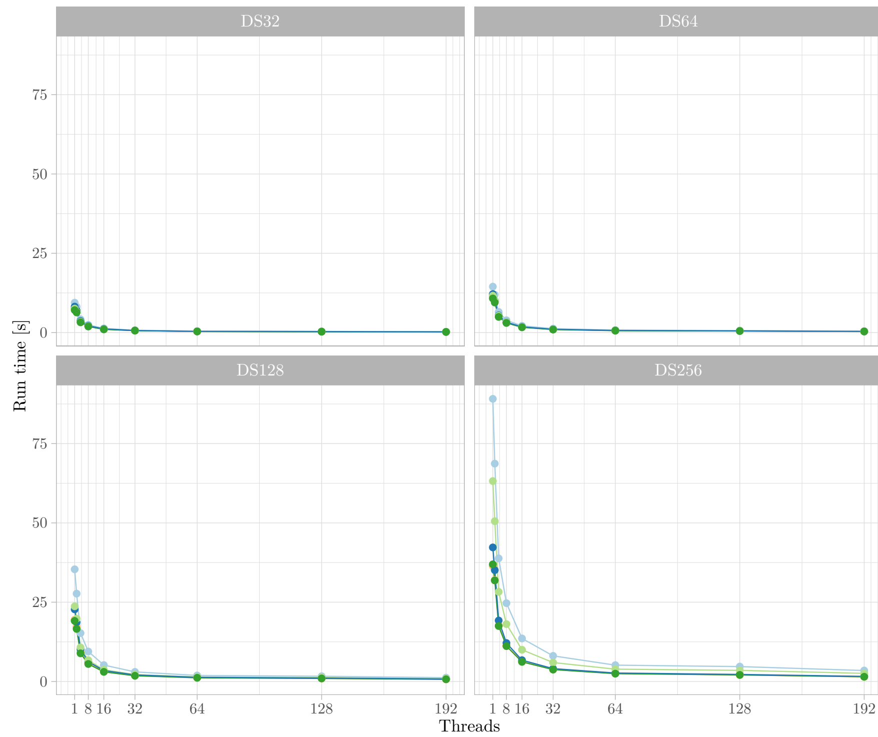
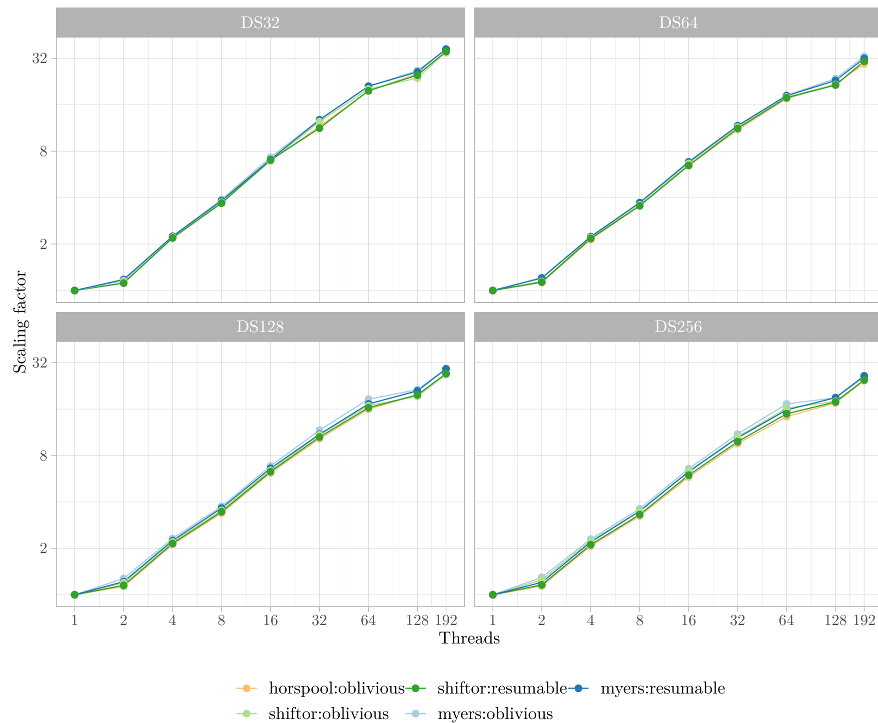


Figure 4.15: Runtimes of online matching algorithms on chr22.

From the results, we can also observe that for more complex algorithms, the performance



(a) Run time comparison.



(b) Scaling comparison.

Figure 4.16: Analysis of multi-threaded performance of JST traversal over chr22 for each of the applied pattern matching algorithms and pattern sequences.

gained from the JST traversal compared to executing the algorithm repetitively on all plain sequences increases. The performance of Myers' algorithm, for example, can be accelerated by a factor of up to 282 times when compared with the plain execution for the DS32 pattern sequence.

Multithreading support

Partitioning the JST allows us to exploit thread-level parallelism on an SMP, as each partial JST can be traversed independently from all other partial JSTs. For this experiment, we split the JST into as many partial JSTs as configured threads and gradually increased the thread count by a factor of 2. The respective run times are depicted in Fig. 4.16a. As can be seen, by adding more threads, the runtime can be greatly reduced, which becomes more significant as the pattern sequences get longer. Moreover, the differences between the pattern matching algorithms become negligible as more threads are added, as they approach the same curve in the plots.

When examining the scaling of the JST traversal, we can observe that up to 64 threads, the parallel traversal scales well with the number of threads, although the scaling factor is less than the number of utilised threads. This can be attributed to the non-uniform distribution of variants across the entire sequence, with some regions having a denser distribution of variants than others. Consequently, the workload is not equally balanced between the threads, resulting in a suboptimal scaling factor. Choosing a more fine-grained partitioning may help overcome these obstacles and improve the scaling factor.

From 64 to 128 threads, the scaling slightly decreases and then starts to improve until 192 threads. This variability can be explained by the fact that the test system only consists of 96 physical cores, and using more concurrently running threads reduces the expected performance gain due to the overhead of scheduling two threads on the same core. However, we can still observe a slight improvement in performance, indicating that hyper-threading can exploit some high-latency operations during the traversal. On the other hand, this observation also suggests that there may be room for improving the overall performance of a JST traversal by reducing the CPI.

4.5.3 Read mapping

In addition to the regular online pattern matching algorithms, we recently extended the pigeonhole filter implemented in SeqAn to enable applications such as read mapping on an RCMS data structure. Our strategy is based on the approach of the read mapper RazerS3 [Weese, 2013], which utilises a q-gram index to identify exact matching seeds between the reads and the reference sequence (refer to Section 2.4.2 for recalling the underlying methods).

To implement this approach as a JST traversal, we utilised the additional components that we recently implemented. These components include the ability to track the currently visited path sequence using our seeking approach, as well as the reverse transformation of the JST. With these features, we first built the q-gram index over the loaded set of reads and configured the pigeonhole filter, selecting the maximal q-gram size q for the given error rate and reads. We then applied the pigeonhole filter by traversing the JST. Whenever the

pigeonhole filter signals a hit for a particular read, we obtained the respective seek position from the currently traversed node and calculated the relative position of the seed on the root path sequence.

Verification

For the verification part, we developed a special JST wrapper class that allows us to extend a path up to a fixed number of total label symbols. The seek position obtained from the pigeonhole filter is used to recover the local JST path where the exact matching seed was found. During the extension process, we apply the resumable Myers' bit-parallel algorithm to first check if the suffix – we assume without loss of generalisability that the suffix is not empty – of the respective read can be aligned to the currently visited JST path within the given maximum error threshold. During the traversal we only considered the optimal candidate for each subtree path for the subsequent prefix extension, if such a candidate exists.

To perform the prefix extension, we utilise the reverse JST implementation. The initial head position of the found seed is mapped to the reversed JST, as the head of the seed must either be in a reference node or in the alternate subtree root of an insertion or SNV variant. This allows us to efficiently instantiate the base node for the prefix extension. With this information, we can apply the same extension algorithm used for the suffix extension by performing a forward traversal on the reversed JST. If the remaining prefix of the read can also be aligned within the remaining error threshold, the reported seek position is translated back into a forward seek position, marking the beginning of the match.

Finally, the reported end position of the suffix extension is combined with the seek position of the prefix extension to obtain a positional description of the final matching region in the JST. With this match description, we can remove duplicates, compute the semi-global alignment between the read and the matching JST region, or handle different mapping modes, such as finding only the best matches, all matches with the optimal error count, or all matches within the specified error threshold.

Performance analysis

We implemented the prototypic read mapping procedure in the `just::map` tool. To evaluate the performance of this method, we used the simulated read dataset DS100x100Ke3 and performed the evaluation with different error rate configurations. The results for single-threaded execution are presented in Table 4.5.

error rate	time [s]
0%	96.18
1%	214.22
2%	355.13
3%	460.13

Table 4.5: Single threaded run times for mapping DS100x100Ke3 to the RCMS of chr22 with different error rates.

With increasing error rates, the overall runtime increases. Using a 3% error rate, the same execution is approximately 4.8 times slower than mapping with zero errors. The significant difference between the various error rates can be attributed to the fact that for higher error rates, the pigeonhole filter considers more seeds per read, resulting in a higher frequency of verification steps and, consequently, longer run times.

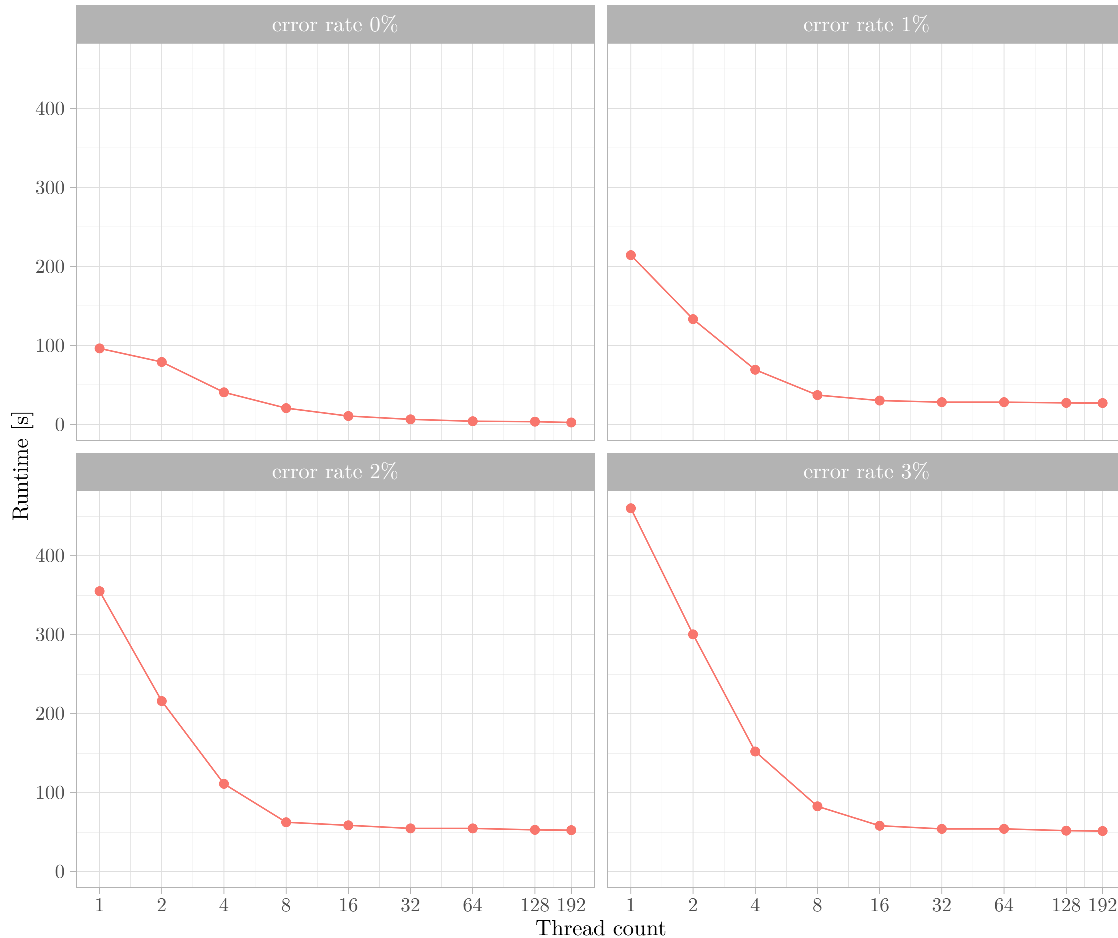


Figure 4.17: Run time comparison for mapping DS100x100Ke3 to the RCMS of chr22 using multiple threads.

In order to improve the performance we can again use multi-threading to increase the IPC. The results are shown in Fig. 4.17. Here, we can observe that the run times in a multi-threaded execution can be reduced to approximately 2, 27, 51, and 51 seconds using error rates of 0%, 1%, 2%, and 3% respectively.

Prefiltration using IBFs

To further improve the performance of the read mapper, we incorporated a coarse counting filter using an IBF (see Section 2.4.2) built over the RCMS of chr22. Our approach involves constructing a partitioned JST, with a total of b bins corresponding to the source sequence.

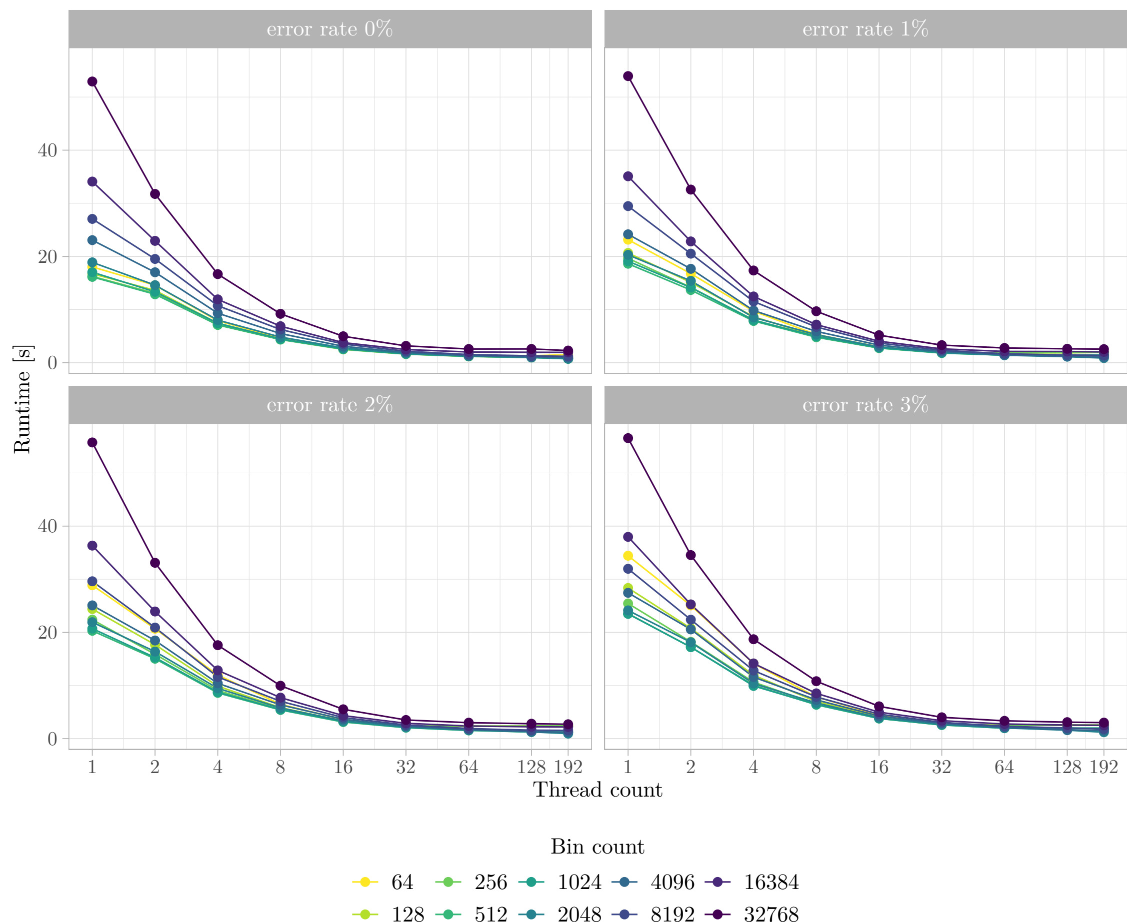


Figure 4.18: Run time comparison for mapping DS100x100Ke3 to the RCMS of chr22 using multiple threads and different bin counts.

Each partial JST represents the sequence content used to construct the corresponding Bloom filters enclosed in the IBF. The IBF itself is constructed with b bins as well.

By leveraging our design, we can conveniently utilise the existing IBF implementation to index the sequence content of a pangenome represented by the partial JST. This is achieved by enumerating consecutively overlapping q-grams in each partial JST and mapping them to positions in the underlying bitvector of the IBF using a set of hash functions. In essence, we employ an online algorithm to construct the IBF, with a window size of $\omega = q$, making it applicable to our JST traversal.

To ensure full sensitivity of the filter, we extend the sequence overlaps of each partial JST to at least $m - 1$, where m is the size of the longest processed read sequence. This prevents the omission of reads located at the border between two adjacent partial JSTs. To achieve this, we adapt the construction of a partial JST by introducing an optional overlap parameter o, added to the end position of the initial root node of each partial JST.

Since the read length is not known beforehand, the user can choose a relatively large overlap

value for the initial construction of the IBF, making it applicable for later experiments with different input data. For example, selecting an overlap of $o = 500$ would suffice to handle almost all short-read sequencing experiments with read lengths up to 500 bases. Although, choosing the overlap too high may increase the number of duplicates with negative effects on the overall performance.

Once constructed, the IBF can be used for the initial coarse filtration stage independently of the actual JST. The IBF already encodes the sequence content of each partial JST. In this stage, we maintain a list of b buckets, one for each counting bin of the IBF, and assign a read Y_j, of an initial read set Y, to the i-th bucket if the counting filter produces a hit for Y_j in bin i. Each produced bucket serves as an independent input for the seed-and-extend approach described earlier that can be executed in parallel. This initial coarse filtration also reduces the size of the q-gram index for each processed bucket improving the performance as well.

The final run times for mapping the reads of DS100x100Ke3, considering different error rates and bin counts, utilising an IBF built for the RCMS of chr22 with a q-gram size of $q = 21$, are presented in Fig. 4.18.

Notably, incorporating an IBF as a prepended coarse filtration step significantly reduces the overall run time. Using a single thread, the performance is improved by a factor of 5.95, 11.49, 17.49, and 19.55 for error rates of 0%, 1%, 2%, and 3%, respectively, using the optimal IBF configuration with 512 bins. Additionally, when employing 64 threads and 512 bins, the mapping of all 100 000 reads is completed in just 1.97 seconds. Furthermore, the figure indicates that with an increased number of threads, the run times for different bin counts converge, suggesting that on highly parallel systems, fewer bins can be utilised while maintaining similar run times.

5 Summary and future work

In this thesis, we focused on two fundamental classes of algorithms essential for a wide range of modern sequence analysis applications: pairwise sequence alignment and pattern matching. We explored various approaches to optimise these algorithms in order to address the significant challenges posed by the massive capacities and throughputs of modern sequencing technologies.

Pairwise Alignment Algorithms

Regarding the computation of pairwise sequence alignments, we investigated different code-level optimisations to enhance the performance of these algorithms by leveraging different levels of parallelism offered by modern CPUs. This included maximising the utilisation of data-level and thread-level parallelisms offered by as well as focussing on optimising cache efficiency and exploiting the instruction-level parallelism of superscalar and superpipelined compute engines.

To achieve this, we interleaved a bulk of DP matrices whose entries can be computed in parallel using SIMD instructions. For this inter-sequence vectorisation layout we investigated more efficient strategies to transform the memory layout of the given input sequences and implemented a saturation mode that enables the full utilisation of all SIMD lanes, regardless of the sequence lengths. Furthermore we developed a striped iteration pattern that allows to unroll the computation of multiple cells on the same row increasing the throughput of the instruction-level parallelism and improving the cache efficiency. In addition, we developed a more efficient solution to handle hetereogenous sequence collections for our vectorisation approach and extended the current solution with an optimised scheme for matrix substitution score functions to improve the performance for aligning amino acid sequences. We further extended this by also adding a special profile configuration that can be employed when aligning a single sequence against a collection of sequences, such as in a database search.

Another significant aspect of our work was the design of a library API aimed at simplifying the usage of optimised alignment algorithms. Here, we explored various design patterns to unify the diverse features of alignment algorithms within a centralised DP algorithm that can be customised through dedicated policy classes.

Lastly, we implemented a dynamic scheduling system to distribute multiple independent alignment instances to different processors of a multiprocessor system. We also developed a flexible task-graph execution model that allows to split the DP matrices of alignment instances into smaller tiles, which can be executed concurrently. The synchronisation is handled automatically by the task dependencies given by the task graph.

Pattern matching algorithms

The second major focus of our research was on pattern matching algorithms. In this context, our objective was to extend the general approach of these algorithms from a single reference sequence to a system comprising thousands of related reference sequences. A crucial requirement was the recognition that these reference systems often consist of similar sequences derived from different individuals within the same population. Incorporating information from an entire population can significantly enhance the quality of analysis results by mitigating undesired effects like reference bias in primary sequence analysis.

However, one of the main challenges in working with these massive datasets is efficient data handling. To address this, we designed and implemented a dedicated data model that stores a collection of similar sequences in a compact form, leveraging a referential sequence encoding strategy. Additionally, we developed a fast sequence representation called the journaled sequence for these encoded sequences. Building upon this approach, we designed and implemented a generalised traversal scheme that utilises journaled sequences to construct a journaled sequence tree (JST) representation from the encoded sequence data. Importantly, our approach is fully dynamic, meaning that the visited paths are dynamically created on-the-fly during the traversal. As a result, our method does not require any additional initialisations or indexing routines and can work with patterns of any size. The general traversal can be further customised using composable tree polishing adapters, allowing for control over the depth of visited alternate subtrees based on parameters such as the minimal window size or path haplotype coverage.

We also designed the operations to facilitate the integration of third-party pattern matching algorithms, originally designed for linear reference sequences, into a pangenomic pipeline. To demonstrate the applicability of our design, we incorporated existing pattern matching algorithms from SeqAn2, including Horspool, ShiftOr, Myers' bitparallel algorithm, and even more complex algorithms like the pigeonhole filter. The extensions we made to our original work, published in 2014, enabled the splitting of encoded data into multiple chunks for independent traversal, reverse traversal, seeking to visited paths using dedicated positions, and improved strategies for projecting JST positions into the coordinate space of each encoded haplotype or computing the haplotype cover of a given JST position.

As part of our efforts, we began building a set of utility tools around our data representation, collectively referred to as the JuST tools. One of these tools is a prototype read mapper that can already map 100 000 reads, each of length 100 bases, to the 1KGP encoded chr22 in less than 2 seconds.

Outlook

Moving forward, our plan is to expand this application into a fully competitive read mapper for pangenomes. This includes the use of canonical minimisers in the coarse filtration step, utilising the recently published hierarchical IBF [Mehringer et al., 2023] to reduce the size of the initial counting index and expedite the initial filtration process, thereby enabling the handling of whole-genome data. We will further integrate our new strategies to efficiently compute pairwise sequence alignments into `just::map`. To provide some perspective, with our improved vectorisation it would be theoretically possible to compute

153 600 000 alignment instances for sequences of length 100 in just one second on a Intel® Xeon® Platinum 8358 Processor when utilising all 64 cores combined with AVX512.

Furthermore, we aim to explore more compact data representations for the encoded data, as the additional gzipped representation of the encoded chromosomes suggests that the data can be stored more efficiently. This particularly pertains to the efficient storage of coverage data.

Similarly, we will focus on optimising the speed of the JST traversal. Profiling the JST traversal revealed that computing the coverage intersection and difference represents the main performance bottleneck. However, not using the haplotype coverage to prune invalid paths would further increase the overall runtime, as the workload considerably expands. Therefore, we plan to vectorise the computation of set intersections based on the idea presented in [Inoue et al., 2014].

Bibliography

Generation Genome, 2016. URL `https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment{_}data/file/631043/CMO{_}annual{_}report{_}generation{_}genome.pdf`.

Abel, A. and Reineke, J. Uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. *Int. Conf. Archit. Support Program. Lang. Oper. Syst. - ASPLOS*, pages 673–686, 2019. doi: 10.1145/3297858.3304062.

Ahmed, N. et al. GASAL2: A GPU accelerated sequence alignment library for high-throughput NGS data. *BMC Bioinformatics*, 20(1), 2019. ISSN 14712105. doi: 10.1186/s12859-019-3086-9.

Alpern, B., Carter, L. and Gatlin, K.S.G.K.S. Microparallelism and High-Performance Protein Matching. *Proc. IEEE/ACM SC95 Conf.*, pages 1–16, 1995. ISSN 10639535. doi: 10.1109/SUPERC.1995.242795.

Auton, A. et al. A global reference for human genetic variation. *Nature*, 526(7571):68–74, oct 2015. ISSN 0028-0836. doi: 10.1038/nature15393. URL `https://www.nature.com/articles/nature15393`.

Baeza-Yates, R.A. Algorithms for String Searching: A Survey. *ACM SIGIR Forum*, 23 (3-4):34–58, 1989. ISSN 01635840. doi: 10.1145/74697.74700.

Baeza-Yates, R.A. and Perleberg, C.H. Fast and practical approximate string matching. *Inf. Process. Lett.*, 59(1):21–27, 1996. ISSN 00200190. doi: 10.1016/0020-0190(96)00083-X.

Bakhvalov, D. *Performance analysis and tuning on modern CPUs.* 2020. URL `https://github.com/dendibakh/perf-book`.

Ballouz, S., Dobin, A. and Gillis, J. Is it time to change the reference genome? *bioRxiv*, pages 1–9, 2019. doi: 10.1101/533166.

Bao, S. et al. Evaluation of next-generation sequencing software in mapping and assembly. *J. Hum. Genet.*, 56(6):406–414, 2011. ISSN 14345161. doi: 10.1038/jhg.2011.43.

Bernardini, G., Pisanti, N., Pissis, S.P. and Rosone, G. Pattern matching on elastic-degenerate text with errors. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 10508 LNCS, pages 74–90, 2017. ISBN 9783319674278. doi: 10.1007/978-3-319-67428-5_7.

Bernardini, G., Pisanti, N., Pissis, S.P. and Rosone, G. Approximate pattern matching on elastic-degenerate text. *Theor. Comput. Sci.*, 812:109–122, apr 2020. ISSN 0304-3975. doi: 10.1016/J.TCS.2019.08.012.

Bernhardsson, C., Wang, X., Eklöf, H. and Ingvarsson, P.K. Variant Calling Using Whole Genome Resequencing and Sequence Capture for Population and Evolutionary Genomic Inferences in Norway Spruce (Picea Abies). pages 9–36, 2020. doi: 10.1007/978-3-030-21001-4_2. URL https://link.springer.com/chapter/10.1007/978-3-030-21001-4{_}2.

Bloom, B.H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. ISSN 15577317. doi: 10.1145/362686.362692.

Bolger, A., Denton, A., Bolger, M. and Usadel, B. LOGAN: A framework for LOssless Graph-based ANalysis of high throughput sequence data. *bioRxiv*, page 175976, 2017. doi: 10.1101/175976.

Brudno, M., Chapman, M., Göttgens, B., Batzoglou, S. and Morgenstern, B. Fast and sensitive multiple alignment of large genomic sequences. *BMC Bioinformatics*, 4:66, dec 2003a. ISSN 1471-2105. doi: 10.1186/1471-2105-4-66. URL http://www.ncbi.nlm.nih.gov/pubmed/14693042http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC521198.

Brudno, M. et al. LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA. *Genome Res.*, 13(4):721–31, apr 2003b. ISSN 1088-9051. doi: 10.1101/gr.926603. URL http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=430158{&}tool=pmcentrez{&}rendertype=abstract.

Burkhardt, S. et al. Q-gram based database searching using a suffix array (QUASAR). *Proc. Annu. Int. Conf. Comput. Mol. Biol. RECOMB*, pages 77–83, 1999. doi: 10.1145/299432.299460.

Cardon, L.R. and Harris, T. Precision medicine, genomics and drug discovery. *Hum. Mol. Genet.*, 25(R2):R166–R172, 2016. ISSN 14602083. doi: 10.1093/hmg/ddw246.

Carrasco-Ramiro, F., Peiró-Pastor, R. and Aguado, B. Human genomics projects and precision medicine. *Gene Ther.*, 24(9):551–561, 2017. ISSN 14765462. doi: 10.1038/gt.2017.77.

Cartwright, R.A. Logarithmic gap costs decrease alignment accuracy. *BMC Bioinformatics*, 7:1–12, 2006. ISSN 14712105. doi: 10.1186/1471-2105-7-527.

Chin, L., Andersen, J.N. and Futreal, P.A. Cancer genomics: From discovery science to personalized medicine. *Nat. Med.*, 17(3):297–303, 2011. ISSN 10788956. doi: 10.1038/nm.2323.

Church, D.M. et al. Modernizing reference genome assemblies. *PLoS Biol.*, 9(7):1–5, 2011. ISSN 15449173. doi: 10.1371/journal.pbio.1001091.

Church, D.M. et al. Extending reference assembly models. *Genome Biol.*, 16(1):2–6, 2015. ISSN 1474760X. doi: 10.1186/s13059-015-0587-3.

Cisłak, A., Grabowski, S. and Holub, J. SOPanG: Online text searching over a pan-genome. *Bioinformatics*, 34(24):4290–4292, 2018. ISSN 14602059. doi: 10.1093/bioinformatics/bty506.

Collins, F.S. and Varmous, H. A new initiative on precision medicine. *N. Engl. J. Med.*, 6 (372):793–795, 2015. ISSN 16640640. doi: 10.1056/NEJMp1500523.

Cookson, C. Genomics promises a leap forward for rare disease diagnosis, 2017.

Cormen, T.H., Stein, C., Leiserson, C.E. and Rivest, R.L. *Introduction to Algorithms*. MIT Press, second edi edition, 1990. ISBN 0-262-03293-7. URL `https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/`.

Culler, D., Singh, J.P. and Gupta, A. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, 1998. ISBN 978-0-08-057307-6.

Dadi, T.H. et al. DREAM-Yara: an exact read mapper for very large databases with short update time. *Bioinformatics*, 34(17):i766–i772, sep 2018. ISSN 1367-4803. doi: 10.1093/bioinformatics/bty567. URL `https://academic.oup.com/bioinformatics/article/34/17/i766/5093228`.

Daily, J. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics*, 17(1):81, 2016. URL `http://www.biomedcentral.com/1471-2105/17/81`.

Dayhoff, M.O., Schwartz, R.M. and Orcutt, B. A model of evolutionary change in proteins. In *Atlas protein Seq. Struct.*, volume 5, pages 345–352. National Biomedical Research Foundation, 1978. URL `https://ci.nii.ac.jp/naid/10024396603`.

Di Resta, C., Galbiati, S., Carrera, P. and Ferrari, M. Next-generation sequencing approach for the diagnosis of human diseases: Open challenges and new opportunities. *Electron. J. Int. Fed. Clin. Chem. Lab. Med.*, 29(1):4–14, 2018. ISSN 16503414.

Döring, A., Weese, D., Rausch, T. and Reinert, K. SeqAn an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, 9:11, jan 2008. ISSN 1471-2105. doi: 10.1186/1471-2105-9-11. URL `http://www.ncbi.nlm.nih.gov/pubmed/18184432`.

Durbin, R. Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272, 2014. ISSN 14602059. doi: 10.1093/bioinformatics/btu014.

Durbin, R., Eddy, S.R., Krogh, A. and Mitchison, G. *Biological sequence analysis : probabalistic models of proteins and nucleic acids*, volume 1. Wageningen University and Research - Library, 1998. ISBN 0521630414.

Edans, E.F. and De Melo, A.C.M.A. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using GPU. *IEEE Trans. Parallel Distrib. Syst.*, 24(5):1009–1021, 2013. ISSN 10459219. doi: 10.1109/TPDS.2012.194.

Edmiston, E.W., Core, N.G., Saltz, J.H. and Smith, R.M. Parallel processing of biological sequence comparison algorithms. *Int. J. Parallel Program.*, 17(3):259–275, 1988. ISSN 08857458. doi: 10.1007/BF02427852.

Emde, A.K. et al. Detecting genomic indel variants with exact breakpoints in single- and paired-end sequencing data using splazers. *Bioinformatics*, 28(5):619–627, 2012. ISSN 13674803. doi: 10.1093/bioinformatics/bts019.

Farrar, M. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–61, jan 2007. ISSN 1367-4811. doi: 10.1093/bioinformatics/btl582. URL `http://www.ncbi.nlm.nih.gov/pubmed/17110365`.

Farrar, M.S. Optimizing Smith-Waterman for the Cell Broadband Engine. (2):1–5, 2008.

Ferlay, J. et al. Cancer statistics for the year 2020: An overview. *Int. J. Cancer*, 149(4):778–789, aug 2021. ISSN 0020-7136. doi: 10.1002/ijc.33588. URL `https://onlinelibrary.wiley.com/doi/10.1002/ijc.33588`.

Ferreira, C.R. The burden of rare diseases. *Am. J. Med. Genet. Part A*, 179(6):885–892, jun 2019. ISSN 1552-4833. doi: 10.1002/AJMG.A.61124. URL `https://onlinelibrary.wiley.com/doi/full/10.1002/ajmg.a.61124https://onlinelibrary.wiley.com/doi/abs/10.1002/ajmg.a.61124https://onlinelibrary.wiley.com/doi/10.1002/ajmg.a.61124`.

Flynn, M.J. Very High-Speed Computing Systems. *Proc. IEEE*, 54(12):1901–1909, 1966. ISSN 15582256. doi: 10.1109/PROC.1966.5273.

Frielingsdorf, J.T. Improving optimal sequence alignments through a SIMD-accelerated library. Technical report, 2015.

Galil, Z. and Giancarlo, R. Data structures and algorithms for approximate string matching. *J. Complex.*, 4(1):33–72, 1988. ISSN 10902708. doi: 10.1016/0885-064X(88)90008-8.

Garrison, E. et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nat. Biotechnol.*, 36(9):875–881, 2018. ISSN 15461696. doi: 10.1038/nbt.4227.

Genomics England et al. The National Genomic Research Library [version 5.1]. 2020. URL `https://doi.org/10.6084/m9.figshare.4530893.v7`.

Gog, S., Beller, T., Moffat, A. and Petri, M. From theory to practice: Plug and play with succinct data structures. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 8504 LNCS:326–337, 2014. ISSN 16113349. doi: 10.1007/978-3-319-07959-2_28.

Goodwin, S., McPherson, J.D. and McCombie, W.R. Coming of age: Ten years of next-generation sequencing technologies. *Nat. Rev. Genet.*, 17(6):333–351, 2016. ISSN 14710064. doi: 10.1038/nrg.2016.49.

Gotoh, O. Optimal sequence alignment allowing for long gaps. *Bull. Math. Biol.*, 52(3):359–373, 1990.

Grossi, R. et al. On-Line Pattern Matching on Similar Texts *. *Leibniz Int. Proc. Informatics, LIPIcs*, 9(9):1–9, 2017. doi: 10.4230/LIPIcs.CPM.2017.9.

Guo, Y. et al. Improvements and impacts of GRCh38 human reference on high throughput sequencing data analysis. *Genomics*, 109(2):83–90, mar 2017. ISSN 10898646. doi: 10.1016/J.YGENO.2017.01.005.

Gurdasani, D., Barroso, I., Zeggini, E. and Sandhu, M.S. Genomics of disease risk in globally diverse populations. *Nat. Rev. Genet.*, 20(9):520–535, 2019. ISSN 14710064. doi: 10.1038/s41576-019-0144-0. URL `http://dx.doi.org/10.1038/s41576-019-0144-0`.

Gusfield, D. Algorithms on strings, trees, and sequences : computer science and computational biology. page 534, 1997.

Harper, A.R. and Topol, E.J. Pharmacogenomics in clinical practice and drug development. *Nat. Biotechnol.*, 30(11):1117–1124, 2012. ISSN 15461696. doi: 10.1038/nbt.2424.

Hauswedell, H., Singer, J. and Reinert, K. Lambda: the local aligner for massive biological data. *Bioinformatics*, 30(17):i349–i355, sep 2014. ISSN 1460-2059. doi: 10.1093/bioinformatics/btu439. URL https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btu439.

Hein, J. A new method that simultaneously aligns and reconstructs ancestral sequences for any number of homologous sequences, when the phylogeny is given. *Mol. Biol. Evol.*, nov 1989. ISSN 1537-1719. doi: 10.1093/oxfordjournals.molbev.a040577. URL https://academic.oup.com/mbe/article/6/6/649/981429/A-new-method-that-simultaneously-aligns-and.

Henikoff, S. and Henikoff, J.G. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. U. S. A.*, 89(22):10915–10919, 1992. ISSN 00278424. doi: 10.1073/pnas.89.22.10915.

Hennessy, J.L. and Patterson, D.A. *textsComputer organization and design : the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Mateo, California, 1994. ISBN 978-1-55-860281-6.

Herbig, A., Jäger, G., Battke, F. and Nieselt, K. GenomeRing: alignment visualization based on SuperGenome coordinates. *Bioinformatics*, 28(12):i7–i15, jun 2012. ISSN 1367-4803. doi: 10.1093/BIOINFORMATICS/BTS217. URL https://academic.oup.com/bioinformatics/article/28/12/i7/268598.

Hickey, G. et al. Genotyping structural variants in pangenome graphs using the vg toolkit. *Genome Biol.*, 21(1):35, dec 2020. ISSN 1474-760X. doi: 10.1186/s13059-020-1941-7. URL https://genomebiology.biomedcentral.com/articles/10.1186/s13059-020-1941-7.

Holley, G. and Melsted, P. Bifrost: Highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biol.*, 21(1):1–20, sep 2020. ISSN 1474760X. doi: 10.1186/S13059-020-02135-8/FIGURES/3. URL https://genomebiology.biomedcentral.com/articles/10.1186/s13059-020-02135-8.

Holtgrewe, M. Mason – A Read Simulator for Second Generation Sequencing Data. *Life Sci.*, (October):18, 2010. URL http://publications.mi.fu-berlin.de/962/{%}5Cnhttp://svn.seqan.de/seqan/trunk/core/apps/mason/README.

Holtgrewe, M. *Engineering Algorithms for Personal Genome Pipelines*. PhD thesis, 2015.

Horspool, R.N. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980. ISSN 1097024X. doi: 10.1002/spe.4380100608.

Huang, L., Popic, V. and Batzoglou, S. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, jul 2013. doi: 10.1093/bioinformatics/btt215. URL http://bioinformatics.oxfordjournals.org/content/29/13/i361.abstract.

Iliopoulos, C., Kundu, R. and Pissis, S. Efficient Pattern Matching in Elastic-Degenerate Strings. pages 1–12, 2016. URL http://arxiv.org/abs/1610.08111.

Inoue, H., Ohara, M. and Taura, K. Faster set intersection with SIMD instructions by reducing branch mispredictions. *Proc. VLDB Endow.*, 8(3):293–304, 2014. ISSN 21508097. doi: 10.14778/2735508.2735518.

Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2023a. URL http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

Intel. Intel® Intrinsics Guide, 2023b. URL https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

Jain, C., Misra, S., Zhang, H., Dilthey, A. and Aluru, S. Accelerating Sequence Alignment to Graphs. In *2019 IEEE Int. Parallel Distrib. Process. Symp.*, pages 451–461. IEEE, may 2019. ISBN 978-1-7281-1246-6. doi: 10.1109/IPDPS.2019.00055. URL https://ieeexplore.ieee.org/document/8821047/.

Jeffers, J., Reinders, J. and Sodani, A. *Intel Xeon Phi Processor High Performance Programming.* 2016. ISBN 0128091940. doi: 10.1016/c2015-0-00549-4.

Jokinen, P. and Ukkonen, E. Two algorithms for approximate string matching in static texts. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 520 LNCS, pages 240–248. Springer Verlag, 1991. ISBN 9783540543459. doi: 10.1007/3-540-54345-7_67. URL https://link.springer.com/chapter/10.1007/3-540-54345-7{_}67.

Kahlert, B. *API Usability of the Template Metaprogramming Software Library SeqAn.* PhD thesis, Freie Universität Berlin, 2015.

Karpiński, P. and McDonald, J. A high-performance portable abstract interface for explicit SIMD vectorization. *Proc. 8th Int. Work. Program. Model. Appl. Multicores Manycores - PMAM'17*, pages 21–28, 2017. doi: 10.1145/3026937.3026939. URL http://dl.acm.org/citation.cfm?doid=3026937.3026939.

Kehr, B., Weese, D. and Reinert, K. STELLAR: fast and exact local alignments. *BMC Bioinformatics*, 12(S9):S15, dec 2011. ISSN 1471-2105. doi: 10.1186/1471-2105-12-S9-S15. URL https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-12-S9-S15.

Kehr, B., Trappe, K., Holtgrewe, M. and Reinert, K. Genome alignment with graph data structures: A comparison. *BMC Bioinformatics*, 15(1), 2014. ISSN 14712105. doi: 10.1186/1471-2105-15-99.

Khajeh-Saeed, A., Poole, S. and Blair Perot, J. Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors. *J. Comput. Phys.*, 229 (11):4247–4258, 2010. ISSN 00219991. doi: 10.1016/j.jcp.2010.02.009. URL http://dx.doi.org/10.1016/j.jcp.2010.02.009.

Khera, A.V. et al. Genome-wide polygenic scores for common diseases identify individuals with risk equivalent to monogenic mutations. *Nat. Genet.*, 50(9):1219–1224, 2018. ISSN 15461718. doi: 10.1038/s41588-018-0183-z. URL `http://dx.doi.org/10.1038/s41588-018-0183-z`.

Kim, D., Paggi, J.M., Park, C., Bennett, C. and Salzberg, S.L. Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype. *Nat. Biotechnol.*, 37 (8):907–915, aug 2019. ISSN 1087-0156. doi: 10.1038/s41587-019-0201-4. URL `https://www.nature.com/articles/s41587-019-0201-4`.

Korpar, M. and Šikić, M. SW#-GPU-enabled exact alignments on genome scale. *Bioinformatics*, 29(19):2494–2495, oct 2013. ISSN 14602059. doi: 10.1093/bioinformatics/btt410. URL `https://academic.oup.com/bioinformatics/article/29/19/2494/2748130`.

Lander, E.S. et al. Initial sequencing and analysis of the human genome. *Nature*, 412 (6846):565–566, 2001. ISSN 0028-0836. doi: 10.1038/35087627.

Lee, C., Grasso, C. and Sharlow, M. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002. URL `http://bioinformatics.oxfordjournals.org/content/18/3/452.short`.

Lee, H. et al. Third-generation sequencing and the future of genomics. *bioRxiv*, (Table 1), apr 2016. doi: 10.1101/048603. URL `http://www.biorxiv.org/content/early/2016/04/13/048603.abstract`.

Leen, W.G. et al. Glucose transporter-1 deficiency syndrome: The expanding clinical and genetic spectrum of a treatable disorder. *Brain*, 133(3):655–670, 2010. ISSN 00068950. doi: 10.1093/brain/awp336.

Leinonen, R., Sugawara, H. and Shumway, M. The sequence read archive. *Nucleic Acids Res.*, 39(SUPPL. 1):2010–2012, 2011. ISSN 03051048. doi: 10.1093/nar/gkq1019.

Leiserson, C.E. et al. There's plenty of room at the top: What will drive computer performance after Moore's law? *Science (80-.).*, 368(6495), 2020. ISSN 10959203. doi: 10.1126/science.aam9744.

Lek, M. et al. Analysis of protein-coding genetic variation in 60,706 humans. *Nature*, 536 (7616):285–291, 2016. ISSN 14764687. doi: 10.1038/nature19057.

Levenshtein, V.I. Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Dokl.*, 10(8):707–710, 1966.

Li, H. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18): 3094–3100, sep 2018. ISSN 1367-4803. doi: 10.1093/bioinformatics/bty191. URL `https://academic.oup.com/bioinformatics/article/34/18/3094/4994778`.

Li, H. and Durbin, R. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, jul 2009. ISSN 1367-4803. doi: 10.1093/BIOINFORMATICS/BTP324. URL `https://dx.doi.org/10.1093/bioinformatics/btp324`.

Li, I.T., Shum, W. and Truong, K. 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). *BMC Bioinformatics*, 8(1):185, 2007. ISSN 14712105. doi: 10.1186/1471-2105-8-185. URL `http://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-8-185`.

Li, J., Ranka, S. and Sahni, S. Pairwise sequence alignment for very long sequences on GPUs. *2012 IEEE 2nd Int. Conf. Comput. Adv. Bio Med. Sci. ICCABS 2012*, 2012a. ISSN 1744-5485. doi: 10.1109/ICCABS.2012.6182641.

Li, Z. et al. Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-bruijn-graph. *Brief. Funct. Genomics*, 11(1):25–37, jan 2012b. ISSN 2041-2649. doi: 10.1093/bfgp/elr035. URL `https://academic.oup.com/bfg/article-lookup/doi/10.1093/bfgp/elr035`.

Liu, Y. and Schmidt, B. SWAPHI: Smith-waterman protein database search on Xeon Phi coprocessors. In *2014 IEEE 25th Int. Conf. Appl. Syst. Archit. Process.*, pages 184–185. IEEE, jun 2014. ISBN 978-1-4799-3609-0. doi: 10.1109/ASAP.2014.6868657. URL `http://ieeexplore.ieee.org/document/6868657/`.

Liu, Y., Wirawan, A. and Schmidt, B. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14(1):117, dec 2013. ISSN 1471-2105. doi: 10.1186/1471-2105-14-117. URL `https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-14-117`.

Liu, Y., Tran, T.T., Lauenroth, F. and Schmidt, B. SWAPHI-LS: Smith-Waterman Algorithm on Xeon Phi coprocessors for Long DNA Sequences. In *2014 IEEE Int. Conf. Clust. Comput.*, number June, pages 257–265. IEEE, sep 2014. ISBN 978-1-4799-5548-0. doi: 10.1109/CLUSTER.2014.6968772. URL `https://ieeexplore.ieee.org/document/6968772`.

Lowy-Gallego, E. et al. Variant calling on the GRCh38 assembly with the data from phase three of the 1000 Genomes Project. *Wellcome Open Res.*, 4:50, dec 2019. doi: 10.12688/WELLCOMEOPENRES.15126.2.

Marco-Sola, S., Moure, J.C., Moreto, M. and Espinosa, A. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics*, pages 1–8, 2020. ISSN 1367-4803. doi: 10.1093/bioinformatics/btaa777.

Marschall, T. et al. Computational pan-genomics: Status, promises and challenges. *Brief. Bioinform.*, 19(1):118–135, oct 2018. ISSN 14774054. doi: 10.1093/bib/bbw089. URL `https://academic.oup.com/bib/article-lookup/doi/10.1093/bib/bbw089`.

Marx, V. A star is born: the updated Human Reference Genome, 2013. URL `http://blogs.nature.com/methagora/2013/12/the{_}updated{_}human{_}reference{_}genome.html`.

McGuire, A.L. et al. The road ahead in genetics and genomics. *Nat. Rev. Genet.*, 21(10):581–596, 2020. ISSN 14710064. doi: 10.1038/s41576-020-0272-6. URL `http://dx.doi.org/10.1038/s41576-020-0272-6`.

Medini, D., Donati, C., Tettelin, H., Masignani, V. and Rappuoli, R. The microbial pan-genome. *Curr. Opin. Genet. Dev.*, 15(6):589–594, 2005. ISSN 0959437X. doi: 10.1016/j.gde.2005.09.006.

Mehringer, S. et al. Hierarchical Interleaved Bloom Filter: Enabling ultrafast, approximate sequence queries. *Genome Biol.*, pages 1–25, 2023. ISSN 1474-760X. doi: 10.1186/s13059-023-02971-4. URL `https://doi.org/10.1101/2022.08.01.502266`.

Metzker, M.L. Sequencing technologies — the next generation. *Nat. Rev. Genet.*, 11(1):31–46, jan 2010. ISSN 1471-0056. doi: 10.1038/nrg2626. URL `https://www.nature.com/articles/nrg2626`.

Muggli, M.D. et al. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017. ISSN 14602059. doi: 10.1093/bioinformatics/btx067.

Muir, P. et al. The real cost of sequencing: Scaling computation to keep pace with data generation. *Genome Biol.*, 17(1):1–9, 2016. ISSN 1474760X. doi: 10.1186/s13059-016-0917-0. URL `http://dx.doi.org/10.1186/s13059-016-0917-0`.

Mulnix, D.L. Intel® Xeon® Processor Scalable Family Technical Overview, 2022. URL `https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html`.

Myers, G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, 1999. ISSN 00045411. doi: 10.1145/316542.316550.

National Institutes of Health. We want to hear from you about changes to NIH's Sequence Read Archive data format and storage, 2020. URL `https://ncbiinsights.ncbi.nlm.nih.gov/2020/06/30/sra-rfi/`.

Navarro, G. Improved approximate pattern matching on hypertext. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 1380, pages 352–357. 1998. doi: 10.1007/BFb0054335. URL `https://link.springer.com/10.1007/BFb0054335`.

Navarro, G. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, mar 2001. ISSN 0360-0300. doi: 10.1145/375360.375365. URL `https://dl.acm.org/doi/10.1145/375360.375365`.

Needleman, S.B. and Wunsch, C.D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48(3):443–453, 1970. ISSN 00222836. doi: 10.1016/0022-2836(70)90057-4. URL `https://www.sciencedirect.com/science/article/pii/0022283670900574`.

Novak, A.M., Garrison, E. and Paten, B. A graph extension of the positional Burrows-Wheeler transform and its applications. *Algorithms Mol. Biol.*, 12(1):1–12, 2017. ISSN 17487188. doi: 10.1186/s13015-017-0109-9.

Oliva, A., Tobler, R., Cooper, A., Llamas, B. and Souilmi, Y. Systematic benchmark of ancient DNA read mapping. *Brief. Bioinform.*, 22(5):1–12, 2021. ISSN 14774054. doi: 10.1093/bib/bbab076.

Ono, Y., Asai, K. and Hamada, M. PBSIM: PacBio reads simulator - Toward accurate genome assembly. *Bioinformatics*, 29(1):119–121, 2013. ISSN 13674803. doi: 10.1093/bioinformatics/bts649.

Pabinger, S. et al. A survey of tools for variant analysis of next-generation genome sequencing data. *Brief. Bioinform.*, 15(2):256–278, 2014. ISSN 14774054. doi: 10.1093/bib/bbs086.

Paten, B., Herrero, J., Beal, K., Fitzgerald, S. and Birney, E. Enredo and Pecan: Genome-wide mammalian consistency-based multiple alignment with paralogs. *Genome Res.*, 18(11):1814–1828, 2008. ISSN 10889051. doi: 10.1101/gr.076554.108. URL `https://genome.cshlp.org/content/18/11/1814.short`.

Paten, B. et al. Cactus: Algorithms for genome multiple sequence alignment. *Genome Res.*, 21(9):1512–1528, sep 2011. ISSN 1088-9051. doi: 10.1101/gr.123356.111. URL `http://genome.cshlp.org/lookup/doi/10.1101/gr.123356.111`.

Paten, B., Novak, A.M., Eizenga, J.M. and Garrison, E. Genome graphs and the evolution of genome inference. *Genome Res.*, 27(5):665–676, may 2017. ISSN 1088-9051. doi: 10.1101/gr.214155.116. URL `http://genome.cshlp.org/lookup/doi/10.1101/gr.214155.116`.

Paten, B. et al. Superbubbles, Ultrabubbles, and Cacti. *J. Comput. Biol.*, 25(7):649, jul 2018. ISSN 10665277. doi: 10.1089/CMB.2017.0251. URL `/pmc/articles/PMC6067107//pmc/articles/PMC6067107/?report=abstracthttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC6067107/`.

Pereira, R., Oliveira, J. and Sousa, M. Bioinformatics and computational tools for next-generation sequencing analysis in clinical genetics. *J. Clin. Med.*, 9(1), 2020. ISSN 20770383. doi: 10.3390/jcm9010132.

Personalized Medicine Coalition. Personalized Medicine Coalition, 2021. URL `http://www.personalizedmedicinecoalition.org/`.

Pevzner, P.A., Tang, H. and Tesler, G. De Novo Repeat Classification and Fragment Assembly. *Genome Res.*, 14(9):1786–1796, sep 2004. ISSN 1088-9051. doi: 10.1101/gr.2395204. URL `http://genome.cshlp.org/lookup/doi/10.1101/gr.2395204`.

Pissis, S.P. and Retha, A. Dictionary matching in elastic-degenerate texts with applications in searching VCF files on-line. *Leibniz Int. Proc. Informatics, LIPIcs*, 103(16):1–14, 2018. ISSN 18688969. doi: 10.4230/LIPIcs.SEA.2018.16.

Pop, M. and Salzberg, S.L. Bioinformatics challenges of new sequencing technology. *Trends Genet.*, 24(3):142–9, mar 2008. ISSN 0168-9525. doi: 10.1016/j.tig.2007.12.006. URL `http://www.ncbi.nlm.nih.gov/pubmed/18262676`.

Qin, D. Next-generation sequencing and its clinical application. *Cancer Biol. Med.*, 16(1):4–10, feb 2019. ISSN 2095-3941. doi: 10.20892/J.ISSN.2095-3941.2018.0055. URL `https://www.cancerbiomed.org/content/16/1/4https://www.cancerbiomed.org/content/16/1/4.abstract`.

Rahn, R., Weese, D. and Reinert, K. Journaled string tree-a scalable data structure for analyzing thousands of similar genomes on your laptop. *Bioinformatics*, 30(24):3499–3505, 2014. ISSN 14602059. doi: 10.1093/bioinformatics/btu438.

Rahn, R. et al. Generic accelerated sequence alignment in SeqAn using vectorization and multi-threading. *Bioinformatics*, 34(20):3437–3445, oct 2018. ISSN 1367-4803. doi: 10.1093/bioinformatics/bty380. URL `https://academic.oup.com/bioinformatics/article/34/20/3437/4992147`.

Rakocevic, G. et al. Fast and accurate genomic analyses using genome graphs. *Nat. Genet.*, 51(2):354–362, feb 2019. ISSN 1061-4036. doi: 10.1038/s41588-018-0316-4. URL `https://www.nature.com/articles/s41588-018-0316-4`.

Ramoni, R.B. et al. The Undiagnosed Diseases Network: Accelerating Discovery about Health and Disease. *Am. J. Hum. Genet.*, 100(2):185–192, 2017. ISSN 15376605. doi: 10.1016/j.ajhg.2017.01.006.

Rasko, D.A. et al. The pangenome structure of Escherichia coli: Comparative genomic analysis of E. coli commensal and pathogenic isolates. *J. Bacteriol.*, 190(20):6881–6893, 2008. ISSN 00219193. doi: 10.1128/JB.00619-08.

Rasmussen, K.R., Stoye, J. and Myers, E.W. Efficient q-gram filters for finding all ϵ-matches over a given length. *J. Comput. Biol.*, 13(2):296–308, 2006. ISSN 10665277. doi: 10.1089/cmb.2006.13.296.

Rausch, T. et al. Segment-based multiple sequence alignment. *Bioinformatics*, 24(16): i187–i192, 2008. URL `http://www.ncbi.nlm.nih.gov/pubmed/18689823`.

Rautiainen, M., Mäkinen, V. and Marschall, T. Bit-parallel sequence-to-graph alignment. *Bioinformatics*, 35(19):3599–3607, 2019. ISSN 14602059. doi: 10.1093/bioinformatics/btz162.

Reinert, K. et al. The SeqAn C++ template library for efficient sequence analysis: A resource for programmers. *J. Biotechnol.*, 261:157–168, nov 2017. ISSN 18734863. doi: 10.1016/j.jbiotec.2017.07.017. URL `https://linkinghub.elsevier.com/retrieve/pii/S0168165617315420`.

Relling, M.V. and Evans, W.E. Pharmacogenomics in the clinic. *Nature*, 526(7573): 343–350, 2015. ISSN 14764687. doi: 10.1038/nature15817.

Roehr, J.T., Dieterich, C. and Reinert, K. Flexbar 3.0 - SIMD and multicore parallelization. *Bioinformatics*, 33(18):2941–2942, 2017. ISSN 14602059. doi: 10.1093/bioinformatics/btx330. URL `https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btx330`.

Rognes, T. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics*, 12(1):221, 2011. ISSN 1471-2105. doi: 10.1186/1471-2105-12-221. URL `http://www.biomedcentral.com/1471-2105/12/221`.

Rognes, T. and Seeberg, E. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8): 699–706, 2000. ISSN 1367-4803. doi: 10.1093/bioinformatics/16.8.699.

Rosenfeld, J.A., Mason, C.E. and Smith, T.M. Limitations of the human reference genome for personalized genomics. *PLoS One*, 7(7), 2012. ISSN 19326203. doi: 10.1371/journal.pone.0040294.

Rucci, E. et al. First Experiences Optimizing Smith-Waterman on Intel's Knights Landing Processor. pages 1–14, 2017. URL `http://arxiv.org/abs/1702.07195`.

Rupp, K. Microprocessor trend data. URL `https://github.com/karlrupp/microprocessor-trend-data`.

Sarje, A. and Aluru, S. Parallel biological sequence alignments on the Cell Broadband Engine. *2008 IEEE Int. Symp. Parallel Distrib. Process.*, (April):1–11, 2008. ISSN 1530-2075. doi: 10.1109/IPDPS.2008.4536328. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4536328`.

Schneider, V.A. et al. Evaluation of GRCh38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly. *Genome Res.*, 27(5):849–864, may 2017. ISSN 15495469. doi: 10.1101/GR.213611.116.

Schork, N.J. Big data: Astronomical or genomical? *PLoS Biol.*, 13(7):1–11, 2015. ISSN 15457885. doi: 10.1371/journal.pbio.1002195. URL `https://www.nature.com/news/personalized-medicine-time-for-one-person-trials-1.17411`.

Seiler, E., Mehringer, S., Darvish, M., Turc, E. and Reinert, K. Raptor: A fast and space-efficient pre-filter for querying very large collections of nucleotide sequences. *iScience*, 24(7):102782, 2021. ISSN 25890042. doi: 10.1016/j.isci.2021.102782. URL `https://doi.org/10.1016/j.isci.2021.102782`.

Shendure, J. and Ji, H. Next-generation DNA sequencing. *Nat. Biotechnol.*, 26(10): 1135–45, oct 2008. ISSN 1546-1696. doi: 10.1038/nbt1486. URL `http://www.ncbi.nlm.nih.gov/pubmed/18846087`.

Sherman, R.M. and Salzberg, S.L. Pan-genomics in the human genome era. *Nat. Rev. Genet.*, 21(4):243–254, 2020. ISSN 14710064. doi: 10.1038/s41576-020-0210-7. URL `http://dx.doi.org/10.1038/s41576-020-0210-7`.

Sherman, R.M. et al. Assembly of a pan-genome from deep sequencing of 910 humans of African descent. *Nat. Genet.*, 51(1):30–35, 2019. ISSN 15461718. doi: 10.1038/s41588-018-0273-y. URL `http://dx.doi.org/10.1038/s41588-018-0273-y`.

Siragusa, E. *Approximate string matching for high-throughput sequencing.* PhD thesis, Freie Universität Berlin, 2015. URL `https://refubium.fu-berlin.de/handle/fub188/11364?show=full`.

Sirén, J., Garrison, E., Novak, A.M., Paten, B. and Durbin, R. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2020. ISSN 14602059. doi: 10.1093/bioinformatics/btz575.

Siriwardena, T.R. and Ranasinghe, D.N. Accelerating global sequence alignment using CUDA compatible multi-core GPU. In *Proc. 2010 5th Int. Conf. Inf. Autom. Sustain. ICIAfS 2010*, pages 201–206, 2010. ISBN 9781424485512. doi: 10.1109/ICIAFS.2010.5715660. URL `https://ieeexplore.ieee.org/abstract/document/5715660/`.

Slatko, B.E., Gardner, A.F. and Ausubel, F.M. Overview of Next-Generation Sequencing Technologies. *Curr. Protoc. Mol. Biol.*, 122(1):1–11, 2018. ISSN 19343647. doi: 10.1002/cpmb.59.

Smedley, D. et al. 100,000 Genomes Pilot on Rare-Disease Diagnosis in Health Care — Preliminary Report. *N. Engl. J. Med.*, 385(20):1868–1880, nov 2021. ISSN 0028-4793. doi: 10.1056/nejmoa2035790.

Smith, T. and Waterman, M. Identification of Common Molecular Subsequences. *J. Mol. Biol.s*, 147:195–197, 1981. ISSN 0022-2836. doi: 10.1016/0022-2836(81)90087-5.

Šoši, M. An SIMD dynamic programming C / C ++ Library. Technical Report no. 741, 2015.

Šošić, M. and Šikić, M. Edlib: A C/C ++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017. ISSN 14602059. doi: 10.1093/bioinformatics/btw753.

Soto-Insuga, V. et al. Glut1 deficiency is a rare but treatable cause of childhood absence epilepsy with atypical features. *Epilepsy Res.*, 154:39–41, aug 2019. ISSN 18726844. doi: 10.1016/j.eplepsyres.2019.04.003.

Spreafico, R., Soriaga, L.B., Grosse, J., Virgin, H.W. and Telenti, A. Advances in genomics for drug development. *Genes (Basel).*, 11(8):1–19, 2020. ISSN 20734425. doi: 10.3390/genes11080942.

Sutter, H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *30(3)*, mar 2005.

Szalkowski, A., Ledergerber, C., Krähenbühl, P. and Dessimoz, C. SWPS3 – fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and ×86/SSE2. *BMC Res. Notes*, 1(1):107, 2008. ISSN 1756-0500. doi: 10.1186/1756-0500-1-107. URL `http://bmcresnotes.biomedcentral.com/articles/10.1186/1756-0500-1-107`.

The Centre for Personalised Medicine. The Centre for Personalised Medicine, 2021. URL `https://cpm.well.ox.ac.uk/`.

Turner, I., Garimella, K.V., Iqbal, Z. and McVean, G. Integrating long-range connectivity information into de Bruijn graphs. *Bioinformatics*, 34(15):2556–2565, aug 2018. ISSN 1367-4803. doi: 10.1093/bioinformatics/bty157. URL `https://academic.oup.com/bioinformatics/article/34/15/2556/4938484`.

Turro, E. et al. Whole-genome sequencing of patients with rare diseases in a national health system. *Nature*, 583(7814):96–102, 2020. ISSN 14764687. doi: 10.1038/s41586-020-2434-2.

Ukkonen, E. Finding approximate patterns in strings, 1985. ISSN 01966774.

Urgese, G. et al. Dynamic Gap Selector: A Smith Waterman Sequence Alignment Algorithm with Affine Gap Model Optimisation. *Proc. IWBBIO*, pages 1347–1358, 2014. URL `http://www.polito.it`.

Venter, J.C. et al. The Sequence of the Human Genome. *Science (80-.).*, 291(5507): 1304–1351, feb 2001. ISSN 0036-8075. doi: 10.1126/science.1058040. URL `https://www.science.org/doi/10.1126/science.1058040`.

Vladimirov, A. A Survey and Benchmarks of Intel® Xeon® Gold and Platinum Processors. pages 1–13, 2017. URL `https://colfaxresearch.com/xeon-2017/https://colfaxresearch.com/download/7107/`.

Wang, D. et al. Glut-1 deficiency syndrome: Clinical, genetic, and therapeutic aspects. *Ann. Neurol.*, 57(1):111–118, 2005. ISSN 03645134. doi: 10.1002/ana.20331.

Wang, Q. et al. Detecting somatic point mutations in cancer genome sequencing data: A comparison of mutation callers. *Genome Med.*, 5(10):1–8, 2013. ISSN 1756994X. doi: 10.1186/gm495.

Wee, Y. et al. The bioinformatics tools for the genome assembly and analysis based on third-generation sequencing. *Brief. Funct. Genomics*, 18(1):1–12, 2019. ISSN 20412657. doi: 10.1093/bfgp/ely037.

Weese, D. Indices and Applications in High-Throughput Sequencing. *PhD Thesis*, 2013. URL `https://refubium.fu-berlin.de/handle/fub188/7664http://publications.mi.fu-berlin.de/1288/`.

Weese, D., Holtgrewe, M. and Reinert, K. RazerS 3: Faster, fully sensitive read mapping. *Bioinformatics*, 28(20):2592–2599, oct 2012. ISSN 1367-4811. doi: 10.1093/bioinformatics/bts505. URL `https://academic.oup.com/bioinformatics/article/28/20/2592/206947`.

Wei, Z. et al. From disease association to risk assessment: An optimistic view from genome-wide association studies on type 1 diabetes. *PLoS Genet.*, 5(10), 2009. ISSN 15537390. doi: 10.1371/journal.pgen.1000678.

Weitzel, J.N., Blazer, K.R., MacDonald, D.J., Culver, J.O. and Offit, K. Genetics, genomics, and cancer risk assessment: State of the Art and Future Directions in the Era of Personalized Medicine. *CA. Cancer J. Clin.*, 61(5):327–59, 2011. ISSN 1542-4863. doi: 10.3322/caac.20128. URL `http://www.ncbi.nlm.nih.gov/pubmed/21858794{%}0Ahttp://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3346864`.

Wozniak, A. Using video-oriented instructions to speed up sequence comparison. *Bioinformatics*, 13(2):145–150, apr 1997. ISSN 1367-4803. doi: 10.1093/bioinformatics/13.2.145. URL `http://bioinformatics.oxfordjournals.org/cgi/content/long/13/2/145`.

Yu, A. The Story of Intel MMX™ Technology. *Intel Technol. J.*, (Q3), 1997. URL `https://www.intel.com/content/dam/www/public/us/en/documents/research/1997-vol01-iss-3-intel-technology-journal.pdf`.

Zhao, M., Lee, W.P., Garrison, E.P. and Marth, G.T. SSW library: An SIMD Smith-Waterman C/C++ library for use in genomic applications. *PLoS One*, 8(12):1–3, 2013. ISSN 19326203. doi: 10.1371/journal.pone.0082138. URL `http://arxiv.org/abs/1208.6350`.

Zheng-Bradley, X. et al. Alignment of 1000 Genomes Project reads to reference assembly GRCh38. *Gigascience*, 6(7):1–8, jul 2017. ISSN 2047217X. doi: 10.1093/GIGASCIENCE/GIX038. URL `https://academic.oup.com/gigascience/article/6/7/gix038/3836916`.

Zook, J.M. et al. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Sci. Data*, 3(1):160025, jun 2016. ISSN 2052-4463. doi: 10.1038/sdata.2016.25. URL `https://www.nature.com/articles/sdata201625`.

List of Figures

1.1 DNA sequencing growth . . . 3

2.1 Alignment example . . . 11
2.2 BLOSUM62 score matrix . . . 15
2.3 Gap score models . . . 16
2.4 Overview of different alignment classes. . . . 17
2.5 DP dependency . . . 17
2.6 DP matrix . . . 18
2.7 Example of computing unbanded and banded alignment . . . 22
2.8 *q*-gram index . . . 27
2.9 Interleaved Bloom Filter . . . 30

3.1 Trends in microarchitecture processor designs . . . 32
3.2 Memory hierarchy and latencies . . . 35
3.3 SIMD register types . . . 38
3.4 Memory access pattern . . . 39
3.5 Amdahl's law. . . . 41
3.6 Parallelised DP matrix computation . . . 46
3.7 Interleaved DP matrix . . . 49
3.8 SIMD transpose . . . 51
3.9 Padded DP matrix . . . 54
3.10 Upper triangular half of the BLOSUM62 substitution score matrix. . . . 57
3.11 Illustration of optimised gather operation . . . 58
3.12 Detailed view of a *b*-tile . . . 61
3.13 Example computing saturated DP matrix . . . 62
3.14 Dependencies of saturated DP tile . . . 63
3.15 Sequence diagram of alignment scheduler and executor . . . 66
3.16 Task graph for tiled DP matrix . . . 69
3.17 Striped DP matrix . . . 71
3.18 Evaluation of task-graph parallelisation . . . 77
3.19 Evaluation of aligning simulated PacBio data . . . 80
3.20 Performance comparison with Parasail . . . 81
3.21 Comparison of new vectorisation using unitary scores . . . 83
3.22 Comparison of new vectorisation using matrix scores . . . 84
3.23 Comparison of new vectorisation using profile scores . . . 85

4.1 Pangenome operations . . . 94
4.2 Plain pangenome . . . 95
4.3 Multiple sequence alignment . . . 96
4.4 Variation graph . . . 97

4.5 Elastic-degenerate sequence . . . 99
4.6 Referentially compressed multisequence . . . 106
4.7 Journaled sequence . . . 114
4.8 Accessing elements in a journaled sequence . . . 115
4.9 Stepwise journaled sequence construction . . . 117
4.10 Position projection . . . 123
4.11 Journaled sequence tree . . . 125
4.12 Referentially compressed multisequence graph . . . 128
4.13 Reversed journaled sequence tree . . . 138
4.14 Chunked referentially compressed multisequence graph . . . 140
4.15 Runtimes of online matching algorithms on chr22. . . . 145
4.16 Evaluation of parallel JST traversal . . . 146
4.17 Evaluation of just::map . . . 149
4.18 Evaluation of just::map with IBF prefilter . . . 150

List of Tables

2.1 Common set notations. 7
2.2 Fundamental set definitions 8
2.3 Common set relations. 8
2.4 Common sequence notations 10

3.1 SIMD instruction set architectures 37
3.2 Classification of alignment requests 45
3.3 Comparing instruction count and throughput of SIMD transpose 52
3.4 Evaluation of vectorised alignment computation 74
3.5 Test data for aligning large-scale sequences 78
3.6 Evaluation of aligning large-scale sequences 79
3.7 Configuration of PBSim. 79
3.8 Performance comparison with Parasail 81
3.9 Alignment algorithm features 88

4.1 Complexity analysis of different dictionary strategies 108
4.2 Complexity analysis of journaled sequence 119
4.3 Summary of transforming VCF files from 1KGP 143
4.4 Evaluation of online pattern matching 145
4.5 Single-threaded just::map run times 148

List of Algorithms

2.1 Naïve search 24

3.1 `AoS2SoA` 50

4.1 `decode` 102
4.2 `decode` from multiencoding 104
4.3 `convert` 111
4.4 `at` 114
4.5 `insert` 115
4.6 `erase` 117
4.7 `replace` 118
4.8 Construct breakpoint tree 120
4.9 `findSupremum` 120
4.10 `search` 121
4.11 `nextRef` 129
4.12 `nextAlt` 130
4.13 `cutAfter` 131
4.14 `cutUncovered` 132
4.15 `expand` 133
4.16 State oblivious JST traversal 134
4.17 Resumable JST traversal 136
4.18 `makeRoot` for partial JST 138
4.19 Wrapped `nextRef` for PJST 139

Index

Symbols

(p,w)-vector 49, 51, 52, 57, 58, 61, 71
de Bruijn graph 98
de novo assembly 93
b-tile 61–65, 69, 174
k-occurrence 24, 26, 27, 29
q-gram index 27
q-gram lemma 29

A

affine gap function 16, 19
aligned symbols 12
alignment executor 66
alignment graph 97
alignment scheduler 66
alternate genome 94
approximate pattern matching 24
array 9
atomic counter 69

B

banded alignment 22
bin 29
Bitap 25
Bloom filter 29
Boyer-Moore-Horspool 25
breakend dictionary 105
breakend supremum 121
breakpoint 101
breakpoint boundary 128
breakpoint span 101
breakpoint tree 121

C

collinear 102
computational pangenomics 94
concatenation 10
concurrent queue 66
condition variable 68
cooptimal alignment 22
core genome 94
counting filter 27
covered sequence difference 103

D

data-level parallelism 33
deletion 13
dictionary 9
differential encoding 96
distributed memory 35
DNA sequencing 93
DP algorithm 19, 26, 50
DP coordinate 19
DP matrix 19, 26

E

edit score function 14
edit scoring scheme 14, 26
effective size 123
elastic-degenerate sequence 99
embarrassingly parallel 41
exact pattern matching 24

F

filter 26
fold expression 73
full-sensitive 27

G

gap 16
gap score function 16
generalised encoding 104
genome graph 98
genome inference 92
genomics 1
graphical pangenome 96

H

haplotype 94

haplotype cover 120
haplotype coverage 104
haplotype position 123
heterogeneous sequence collection 51
homogeneous sequence collection 51
Horspool 25

I

InDel dictionary 105, 106, 109
index 24
insertion 13
instruction-level parallelism 33
inter-sequence 46
interleaved score 49
intra-sequence 46

J

journal 114
journal entry 114

L

latency 34
lexicographic order 11
lexicographic rank 28
linear gap function 16, 17, 19
lossless 27

M

match 13
matrix 9
matrix score function 15
memory hierarchy 33
minor diagonal 48
mismatch 13
multiple sequence alignment 12, 97

O

online pattern matching 24
optimal alignment 17, 18
optimal global alignment 17–19
optimal local alignment 17, 20
optimal overlap alignment 17, 20
optimal semi-global alignment 17, 20

P

padding symbol 54
pairwise sequence alignment 12, 13
pairwise sequence alignment 4, 12, 13
pangenome 94
partial journaled sequence tree 139
pattern matching algorithm 4, 24
pigeonhole filter 27
pigeonhole principle 27
precision medicine 1
predicate 24
prefix 11
primary sequence analysis 92
process 41
profile score function 57
proper differential encoding 102

Q

query sequence 12

R

rank 10
read 93
reference bias 94
reference genome 93
reference sequence 12, 24
reference-centric pangenome 96
referential sequence compression 101
resumable traversal 134

S

saturated b-tile 62
saturated execution 60
secondary sequence analysis 92
seeds 27
segment 11
sensitivity 27
SeqAn 4
sequence 10
sequence difference 101
sequence graph 97
shared memory 35
ShiftOR 25
simultaneous multi-threading 43
spatial locality 37
specificity 26
state-oblivious traversal 134
substitution 13
substitution score function 15
suffix 11
superpipelined 43
superscalar 43

T

target sequence 101
task 41
task graph 69
temporal locality 37
thread 41
thread pool 66
thread-level parallelism 33
thread-local 41, 71
throughput 34
tiled DP matrix 61
trace matrix 21

U

unitary score function 15

V

variant analysis 92
variation graph 98
vector lane 38

Abbreviations

Notation	Description
μop	micro-operation
100KGP	100 000 Genomes Project
1KGP	1000 Genomes Project
AI	alignment instance
ALU	arithmetic logic unit
API	application programming interface
AT	alignment task
AVX2	Advanced Vector Extensions 2
AVX512	Advanced Vector Extensions 512
BF	Bloom Filter
BLOSUM	blocks substitution matrix
CPI	cycles per instruction
CPU	central processing unit
CT	cycle time
DAG	directed acyclic graph
DNA	deoxyribonucleic acid
DP	dynamic programming
ExAC	Exome Aggregation Consortium
FPGA	field-programmable gate array
GCUPS	Giga Cell Updates Per Second
GPU	graphics processing unit
IBF	Interleaved Bloom Filter
ILP	instruction-level parallelism
InDel	Insertion, Deletion
IPC	instructions per cycle
ISA	instruction set architecture
IUPAC	International Union of Pure and Applied Chemistry
JS	journaled sequence
JST	journaled sequence tree
LoBA	Long, Bulk Alignment
LoSA	Long, Single Alignment
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MPMC	multiple producer, multiple consumer
NGS	next-generation sequencing
OOO	out-of-order
OS	operating system

Notation	Description
PAM	point accepted mutation matrix
PJST	partial journaled sequence tree
RCMS	referentially compressed multisequence
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SMP	shared memory multiprocessor
SNV	single nucleotide variant
SoBA	Short, Bulk Alignment
SoSA	Short, Single Alignment
SRA	Sequencing Read Archive
SSE4	Streaming SIMD Extensions 4
TCGA	The Cancer Genome Atlas
TLP	thread-level parallelism
VCF	variant calling format
VPU	vector perocessing unit

Selbstständigkeitserklärung

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Dissertation selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht. Diese Dissertation wurde in gleicher oder ähnlicher Form noch in keinem früheren Promotionsverfahren eingereicht. Mit einer Prüfung meiner Arbeit durch ein Plagiatsprüfungsprogramm erkläre ich mich einverstanden.

Berlin 29.06.2023 ______________________

René Rahn