

High Performance Computing for Genomics



Esteban Pérez Wohlfeil

Departamento de Arquitectura de Computadores
Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

Tesis Doctoral

Tecnologías Informáticas
Aplicaciones y arquitecturas paralelas

Director: Dr. Nicolás Guil Mata
Tutor: Dr. Eladio Gutiérrez Carrasco

Marzo 2023



UNIVERSIDAD
DE MÁLAGA

AUTOR: Esteban Pérez Wohlfeil

 <https://orcid.org/0000-0002-3572-686X>

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga



Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional:

<http://creativecommons.org/licenses/by-nc-nd/4.0/legalcode>

Cualquier parte de esta obra se puede reproducir sin autorización

pero con el reconocimiento y atribución de los autores.

No se puede hacer uso comercial de la obra y no se puede alterar, transformar o hacer obras derivadas.

Esta Tesis Doctoral está depositada en el Repositorio Institucional de la Universidad de Málaga (RIUMA): riuma.uma.es



DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D./Dña ESTEBAN PÉREZ WOHLFEIL

Estudiante del programa de doctorado TECNOLOGÍAS INFORMÁTICAS de la Universidad de Málaga, autor/a de la tesis, presentada para la obtención del título de doctor por la Universidad de Málaga, titulada: COMPUTACIÓN DE ALTO RENDIMIENTO PARA LA GENÓMICA

Realizada bajo la tutorización de ELADIO GUTIÉRREZ CARRASCO y dirección de NICOLÁS GUIL MATA (si tuviera varios directores deberá hacer constar el nombre de todos)

DECLARO QUE:

La tesis presentada es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, conforme al ordenamiento jurídico vigente (Real Decreto Legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), modificado por la Ley 2/2019, de 1 de marzo.

Igualmente asumo, ante a la Universidad de Málaga y ante cualquier otra instancia, la responsabilidad que pudiera derivarse en caso de plagio de contenidos en la tesis presentada, conforme al ordenamiento jurídico vigente.

En Málaga, a 15 de MARZO de 2023

Fdo.: ESTEBAN PÉREZ WOHLFEIL Doctorando/a	Fdo.: ELADIO GUTIÉRREZ CARRASCO Tutor/a
Fdo.: NICOLÁS GUIL MATA Director/es de tesis	

Declaración

Dr. Nicolás Guil Mata
Profesor Catedrático del Departamento de
Arquitectura de Computadores de la
Universidad de Málaga.

Dr. Eladio Gutiérrez Carrasco
Profesor Titular del Departamento de
Arquitectura de Computadores de la
Universidad de Málaga.

CERTIFICAN:

Que la memoria titulada "High Performance Computing for Genomics" ha sido realizada por D. Esteban Pérez Wohlfeil bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y constituye la Tesis que presenta para optar al grado de Doctor en Ingeniería Informática.

Málaga, a 15 de Marzo de 2023.

Autorización para la lectura de la Tesis

Los abajo firmantes declaran, bajo su responsabilidad, que autorizan la lectura de la tesis del doctorando D. Esteban Pérez Wohlfeil, titulada "High Performance Computing for Genomics".

Málaga, a 15 de Marzo de 2023.

Informe de la utilización de las publicaciones que avalan la Tesis Doctoral

El Dr. Nicolás Guil Mata, director de la tesis doctoral de D. Esteban Pérez Wohlfeil, titulada "High Performance Computing for Genomics", declara que ninguna de las publicaciones que avalan dicha tesis han sido utilizadas en tesis doctorales anteriores.

Málaga, a 15 de Marzo de 2023.

I would like to dedicate this thesis to my friend and former PhD Advisor, Prof. Oswaldo Trelles. Wherever he is now, I am sure he is having a great time.

Acknowledgements

This work is the outcome of a learning process that has spanned several years, and I am certain it would not have been possible without the help of the many people I would like to thank.

First I want to thank my family for awakening in me the interest for science, particularly my sister Vega for always having an answer when I asked *why*, my parents Marie and Manuel for their support and for allowing me to develop my interest in computers from an early age, and my life partner Annette for all the support, especially for listening to all my presentation rehearsals throughout the years and for coping with all the times that I was absent. I cannot forget to thank my friend Felipe, for it was due to him that I rejoined my undergraduate program in Computer Science after dropping out.

I would also like to express my gratitude to all of my coworkers, especially Sergio Díaz, with whom I spent countless days working together; Eugenia Ulzurrun for always looking after us; Óscar Torreño for mentoring and guiding me; Jose Arjona for sharing his inspiring creativity and enthusiasm; and of course my friends Marcos Sepúlveda, Ricardo Holthausen, Pablo Brazzarola and all of my colleagues at the 2.3.10 office.

I must also thank all my coworkers at the Departamento de Arquitectura de Computadores at the University of Málaga, especially Dr. Nicolás Guil and Dr. Eladio Gutiérrez for taking over the supervision of this thesis during its final stage. Lastly, I want to thank everyone at the Instituto Nacional de Bioinformática for all these years of collaborations, my international colleagues at the ELIXIR platform and Prof. Sepp Hochreiter and all of the team at the Institute of Machine Learning of the Johannes Kepler University for giving me the opportunity to spend three months learning from them.

I am also grateful for all the financial support received from Instituto de Salud Carlos III (PT13.0001.0012, RD07/0064/0017, RD12/0013/0006 and PT17.0009.0022), the Instituto Nacional de Bioinformática, the Programa Operativo FEDER from the Junta de Andalucía 2014-2020, the European Commission funded project Mr.Symbiomath (code 324554), the ELIXIR-EXCELERATE platform (grant no. 676559 and INFRADEV- 1-H2020 Code 676559), the "Plan Propio" of the University of Málaga and lastly the Instituto de Investigación Biomédica de Málaga (IBIMA).

However and above all, I want to wholeheartedly thank my friend and PhD advisor, Prof. Oswaldo Trelles. He took me in when I had basically no working experience, and trained me with patience and dedication. He always persuaded me to go in the right direction to improve and develop both academically and as a person. I will always be in debt for everything that he taught me, and for all the experiences that I have been able to live and enjoy as a result.

As he would probably phrase it after such a "long" acknowledgement section: "*¡A trabajar, carajo!*"

Abstract

With the thrive of new data acquisition methods, computerized research has become increasingly more common throughout the last decades. However, in order to match the huge data-processing demands, the design of new algorithms along with their optimization on specific hardware platforms (such as Graphic Processing Units) has become a necessity, especially given the costs of running and maintaining large supercomputing infrastructure, both in the case of dedicated and cloud computing servers. This scenario is particularly true in the case of comparative genomics, where massive DNA sequences are being published on a daily basis, and their processing presents a series of computational bottlenecks. As one of the principal use cases, researchers need to calculate alignments between pairs of sequences –which can be generalized as a multiple Longest Common Subsequence problem– with the aim of determining structurally similar regions.

On one hand, the comparison of DNA sequences is a central problem to many applications with direct impact on human health, and therefore its computational acceleration is of wide interest to both the scientific community and humanity as a whole. On the other hand, due to its arbitrary nature, the parallel acceleration of sequence comparison poses interesting computational challenges, including heterogeneous granularity, unpredictable load, balancing and synchronization mechanisms, *etc.* Therefore, in order to achieve high performance, algorithms must be tailored to the appropriate underlying hardware processing model, which may represent completely different computational approaches (*e.g.* transforming task parallelism into data parallelism) and often require even the complete redesign of the algorithms themselves.

This thesis addresses a computational tour of the sequence comparison problem by making use of hardware and algorithmic optimizations in single core machines, shared memory systems and Graphic Processing Units. In particular, the first contribution features a statistical and mathematical framework that enables virtually unlimited search space size in sequence comparison due to its novel hashing methodology and strictly $O(n)$ time complexity. The second contribution describes a parallel approach using shared memory machines to achieve

high sensitivity in the comparison of noisy and partially incomplete metagenomic sequences. The third contribution describes the overcoming of the limitations of the data parallelism model in GPUs for the exhaustive and irregular pairwise sequence comparison. Lastly, a contribution which combines all of the previous knowledge is presented, featuring the use of Machine-Learning-aided schedulers to improve resource allocation and throughput in supercomputers dedicated to sequence comparison.

The direct results of this thesis are twofold: from a computational perspective, new High Performance Computing methodologies, data structures and parallel mechanisms are proposed for a variety of hardware architectures; and from a comparative genomics perspective, the complexity of the sequence comparison problem has been effectively lowered while providing exhaustive and heuristic approaches that can be run on both commodity and specialized hardware.

Table of contents

List of figures	xvii
List of tables	xix
1 Introduction	1
1.1 Research objectives	3
1.2 Contributions	4
1.3 Outline of this thesis	5
2 Background	7
2.1 Central Processing Units, Shared-memory Machines and Graphic Processing Units	9
2.2 Acceleration potential by hardware architecture	10
2.3 Sequence comparison	12
2.3.1 Definitions	13
2.3.2 Differences with common string matching	14
2.3.3 Algorithms	19
2.4 Machine Learning	27
2.4.1 Decision trees	27
2.4.2 Gradient boosting trees	28
2.4.3 Neural networks	29
3 Ultra-fast sequence comparison on single-processor systems	31
3.1 Methods	31
3.1.1 Introduction	31
3.1.2 Formulation	32
3.1.3 Algorithm overview	33
3.1.4 Indexing	33
3.1.5 Matching	35

3.1.6	Selection of local optima	36
3.1.7	Scoring of comparisons	37
3.1.8	Detection of evolutionary events	39
3.1.9	Pipelined and streaming algorithm	40
3.2	Results	41
3.2.1	Introduction	41
3.2.2	Datasets	41
3.2.3	Runtime and memory comparison	42
3.2.4	Coverage and quality comparison	44
3.2.5	Massive comparison of complete genomes	47
4	Shared-memory multiprocessing for the fine-grained alignment of reads	49
4.1	Methods	49
4.1.1	Introduction	49
4.1.2	Algorithm overview	50
4.1.3	Indexing	52
4.1.4	Multiple-stage alignment process	53
4.1.5	Word match and extension policy	53
4.1.6	Bounded Needleman-Wunsch	54
4.1.7	Parallel implementation	55
4.2	Results	58
4.2.1	Introduction	58
4.2.2	Datasets	59
4.2.3	Sequential optimization comparison	59
4.2.4	Parallel evaluation	60
4.2.5	State of the art comparison	61
4.2.6	Comparison assessment, bias and alignment results	65
5	GPU-acceleration for the irregular and exhaustive sequence comparison	67
5.1	Methods	67
5.1.1	Introduction	67
5.1.2	The original GECKO algorithm	68
5.1.3	Subsequence batching	71
5.1.4	Pinned memory	73
5.1.5	Recomputation in subsequence comparisons	73
5.1.6	Algorithm stages and kernel implementations	74
5.2	Results	95

5.2.1	Introduction	95
5.2.2	Infrastructure	95
5.2.3	Datasets	96
5.2.4	Runtime comparison with the CPU algorithm	96
5.2.5	State of the art comparison	99
5.2.6	Comparison using different Graphics Processing Units	102
5.2.7	Runtime complexity analysis	104
5.2.8	Large-scale genome comparison study	107
6	Scheduling of genome comparisons using Machine Learning	111
6.1	Methods	111
6.1.1	Introduction	111
6.1.2	Formulation	112
6.1.3	Overview of the methodology and datasets	114
6.1.4	Comparisons with GPUGECKO	116
6.1.5	Data preprocessing	116
6.1.6	Normalization	118
6.1.7	Evaluated models	119
6.1.8	Scheduling simulation	120
6.2	Results	123
6.2.1	Introduction	123
6.2.2	Infrastructure	123
6.2.3	Training and validation comparison of the models	123
6.2.4	Precision and recall	127
6.2.5	Supercomputer scheduling simulation	130
6.2.6	Explaining predictions via contribution analysis	132
6.2.7	Predictions as a regression problem	133
7	Conclusions and future work	137
7.1	Conclusions	137
7.2	Future work	141
7.3	Additional work	142
7.3.1	Additional conferences and publications	142
7.3.2	Workshops	144
7.3.3	International research stay	144
	References	145

Appendix A	Hardware differences for exploiting performance in CPUs, GPUs and FPGAs	157
A.1	Central Processing Unit (CPU)	157
A.1.1	Architecture overview	158
A.2	Graphic Process Unit (GPU)	161
A.2.1	Compute model	162
A.2.2	Memory model	163
A.2.3	Evolution of GPUs	165
A.3	Field Programmable Gate Arrays	166
A.3.1	Architecture overview	167
Appendix B	Sequential computation of k-mers	171
Appendix C	Custom memory pool allocator in ModernGPU merge sort	173
Appendix D	List of publications supporting this thesis.	177
D.1	Publication 1	177
D.1.1	Abstract	177
D.2	Publication 2	179
D.2.1	Abstract	179
D.3	Publication 3	180
D.3.1	Abstract	180
Appendix E	Resumen en Español - Summary in Spanish	181
E.1	Introducción	181
E.2	Primera contribución	183
E.3	Segunda contribución	185
E.4	Tercera contribución	186
E.5	Cuarta contribución	188
E.6	Conclusiones y trabajo futuro	189

List of figures

2.1	Architecture differences between CPUs and GPUs	11
2.2	Growth of matching words (seeds) per linear increment of the input	17
2.3	Tail inspection of the growth of matching words (seeds)	18
2.4	Distribution of High-scoring Segment Pairs by length	18
2.5	Dotplot comparison between coagulation factors and the SARS-COV2 virus	20
3.1	Workflow diagram for the ultra-fast processing of sequence comparisons on single-processors	34
3.2	Local optima selection from compressed matrix to marginal optima	37
3.3	Chromosome scoring metric examples	38
3.4	Pipelined subsequence streaming algorithm	41
3.5	Similarity map between species	48
4.1	Workflow diagram of the internals of the shared-memory acceleration of metagenomic comparisons	51
4.2	Bounded Needleman-Wunsch optimizations	54
4.3	Custom scheduler and partitioning scheme	58
4.4	Speedup of the shared-memory aligner	62
4.5	Efficiency of the shared-memory aligner	62
5.1	Subsequence batching procedure	72
5.2	Workflow diagram of the GPU-accelerated sequence comparison	75
5.3	State of the art comparison of the GPU accelerated method	100
5.4	State of the art comparison of the GPU accelerated method (II)	101
5.5	Runtime comparison by hardware	103
5.6	Speedup comparison by hardware	104
5.7	Complexity of GPUGECKO and GBLASTN	105
5.8	Linear fit between seeds and runtime	106
5.9	Complexity of GBLASTN	108

5.10	Coverage heatmap of the human and mouse comparison	109
5.11	Coverage heatmap of the human and gorilla comparison	110
6.1	Trinucleotide frequencies in several species	117
6.2	Residual neural network architecture	121
6.3	Job submission distribution of the CIEMAT supercomputer	122
6.4	Learning curve for the Random forest model	124
6.5	Log loss curve for the Gradient boosting model	124
6.6	Learning curve for the mammalian dataset	126
6.7	Learning curve for the aves dataset	126
6.8	Learning curve for the fish dataset	127
6.9	Validation errors by target predictions	129
6.10	Throughput simulation	131
6.11	SHAP explanations of the neural network	132
6.12	Validation of the regression model	134
A.1	Architecture of a CPU	159
A.2	Architecture of a GPU	161
A.3	Architecture of an FPGA	168

List of tables

3.1	Runtime and memory comparison between CHROMEISTER and state-of-the-art methods	43
3.2	Runtime and memory comparison between the pipelined algorithm and state-of-the-art methods	44
3.3	Coverage analysis of the pipelined method	44
3.4	Effect of the inexact word matching heuristic	45
3.5	Effect of the uniqueness in word matching	47
3.6	Effect of the uniqueness in final alignments	47
4.1	Summary of the length distributions in the metagenomic comparisons . . .	60
4.2	Runtime and speedup comparison for the shared-memory parallel aligner with and without optimizations	60
4.3	Parallel speedup and efficiency for the shared-memory aligner	61
4.4	Metagenomic state-of-the-art comparison	63
4.5	Parallel metagenomic state-of-the-art comparison	65
5.1	Runtime analysis of GECKO	70
5.2	Kernel properties for the computation of k -mers	77
5.3	Number of seeds per pairwise comparison	81
5.4	Non-static partitioning of High-scoring Segment Pairs	93
5.5	Speedup comparison between the CPU and GPU algorithms (I)	97
5.6	Speedup comparison between the CPU and GPU algorithms (II)	97
5.7	Exhaustive coverage comparison	101
5.8	Heuristic coverage comparison	102
6.1	Genome species in the training dataset	115
6.2	Genome species in the validation dataset	115
6.3	Validation sets by level of familiarity	118
6.4	Combined probabilities for dinucleotide frequencies	119

6.5	Hyperparameters for Random Forest and Xgboost	120
6.6	Accuracy comparison between models	125
6.7	Confusion matrix of the neural network	128
6.8	Makespan of the throughput simulation	131
C.1	Real and system time comparison without custom memory allocation . . .	173
C.2	Real and system time comparison with custom memory allocation	174

Chapter 1

Introduction

Nowadays, scientific research is most likely to be aided by a computer. From small to increasingly large and complex problems, computers are used to perform several of the tasks that were previously reserved for humans. This includes from archiving data to virtually any type of processing, such as the aggregation of information or the inference or mining of patterns. The ever-growing dependency on computerized research is typically addressed as scientific computing, which can be defined as a compendium of algorithms, computing hardware and the supporting physical infrastructure required for deployment. While the latter field is closer to traditional industrial engineering sciences (dealing with physical problems such as power provision or cooling capabilities, *etc.*), the first two, namely algorithms and computer hardware, often go hand in hand in regards to their goal: achieving both the highest possible computational performance and efficiency at the same time.

From a computer hardware perspective, numerous architectures and processing paradigms have been proposed and used over time. In particular, supercomputers are the common choice for massive and complex scientific experiments, but recently accelerators such as Graphics Processing Units (GPUs) or Field-programmable Gate Arrays (FPGAs) have come to join forces with both large supercomputers and smaller computer infrastructure. The latter, typically being more available to individual research groups, has benefited enormously from GPUs and FPGAs by enabling massive calculation capabilities while keeping a controlled budget. However, as it often occurs, the benefits in one aspect come with disadvantages in others. In this case, the increasing difficulty of fully taking advantage of these acceleration devices (as opposed to that of simpler-but-larger distributed systems) requires the design of ingenious algorithms which combine both computer architecture and domain-specific knowledge. For instance, coarse-grained parallelization applied to supercomputers can be as simple in some scenarios as partitioning the data in batches and distributing them among

independent processors, whereas in the case of *e.g.* GPUs, other factors need to be considered more deeply, such as the memory access patterns (coalescence) or the ratio between memory and compute utilization.

Furthermore, while algorithms can be generalized and extrapolated to different fields and applications, some dependency on the underlying distributions of the data to be processed is always expected. Such data dependency implies that a previous exploration of the application domain is required and, in fact, a better understanding of the boundaries of data enables researchers to design algorithms with increased performance, reliability and overall robustness. This is the case of Comparative Genomics, an effervescent field currently overwhelmed by the availability and exponential growth of data repositories which is causing a shift from the traditional data bottleneck towards a processing bottleneck. In particular, the increasing size and number of genomic sequences (such as those of mammals and plants) are helping researchers to complete their understanding of the evolution of species, which is a crucial problem with direct impact on human health, agriculture, farming, *etc.* However, due to the intrinsic complexity of genomic information, current state-of-the-art methods are being left behind in terms of processing speed, memory footprint, exploitation of parallelism, *etc.*, and new algorithms and parallelization mechanisms able to compensate for the huge rates of data acquisition are required.

Throughout this thesis, new High Performance Computing algorithms applied to Comparative Genomics will be proposed, reviewed and discussed, while incorporating domain-specific knowledge and focusing on computational theory, computer architecture and hardware acceleration. Nevertheless, these findings can be extrapolated to other fields related to regular string search, such as pattern matching, word counting, text corpus comparison and diffing, database search, *etc.*, and a wide variety of fields that are either fully or partially inheriting from the Longest Common Subsequence problem (or LCS in advance). In fact, the case of sequence alignment can be generalized as a multidimensional variant of the LCS problem where rather than searching for the single longest shared substring, all substrings of variable size (for a minimum threshold) need to be found while also allowing for constrained inexactitude, flexibility and biological sense. Moreover, and despite all efforts regarding heuristic methodologies –which are reviewed in section 2.3 "Sequence comparison"–, the finding of all subpatterns (even with constraints and minimum thresholds) is still a fundamentally and inevitable $O(n^2)$ problem in terms of the input.

This fact is a direct consequence of the growth of the search space when comparing two strings. As an example, consider the most simplistic scenario where two strings composed

of only one character are being compared on a per-character basis (*e.g.* AAA and AA): the output number of character matches is the product of the lengths of both sequences $n * m$, since each letter matches at every position of the opposite string. The same principle applies to longer substrings, however, practical applications will end up parameterizing and limiting the minimum size of substrings in order to cope with the quadratic output space. In any case, the search for optimality is completely unfeasible without sufficient computational power and specifically-tailored algorithms.

The big picture of this thesis addresses both the leveraging of computation in the sequence comparison problem with heuristic methodologies for single processors as well as with exhaustive, efficient and optimal substring search algorithms that exploit the massive parallelism available in modern Graphics Processing Units and shared-memory machines. This manuscript is organized as follows: First, a brief section describing the research objectives and contributions will be included, followed by a review of the state of the art of the related fields (particularly computer architecture and sequence comparison). Second, an incremental tour of the methods will be proposed for different hardware infrastructures, namely (1) a heuristic and ultra-fast sequence comparison algorithm for single-processors, (2) a parallel method for the comparison of metagenomes using shared-memory machines and (3) the acceleration of irregular seed-and-extend sequence comparison using Graphics Processing Units. Third, the experimental evaluation will be presented for each of the proposed methods along with discussion. Fourth, the conclusions of the work proposed in this thesis will be described and followed by potential future work. Lastly, additional annexes will be also available including and extending details that were left out of the main manuscript for the sake of clarity.

1.1 Research objectives

The expected objectives set for this thesis are multidisciplinary. Therefore, three main fields have received attention and have been affected by the development of this thesis, namely (1) computer science algorithms, (2) computer architecture and acceleration and (3) comparative genomics, with particular emphasis in sequence comparison. The list of objectives can be summarized per each field as follows:

From the perspective of computer science:

- The study and description of a formal algorithmic theory that supports and unifies each of the architectural and implementation decisions throughout the scientific contributions.
- A contribution to the current state-of-the-art in the form of an abstract review describing the nature of the informational and computational processes that take place in comparative genomics.

From the perspective of computer architecture technologies:

- The review and understanding of the existing landscape of computer architecture.
- The development of new models and schemes for the application of both High Performance Computing and parallel techniques to a wide range of machines and acceleration devices.

From a computational biology perspective:

- The effective reduction of the computational complexity of genome comparison throughout the proposal of new algorithms and their hardware acceleration.

The above-mentioned compendium of objectives represents the materialization of the research carried out throughout the duration of this thesis, whose impact is transversal: (1) in a theoretical approach (formal definitions and parallelization schemes), (2) in a practical approach (technologies, architecture optimization), and lastly (3) in the deployment of production-ready software and its integration and adoption in large communities and platforms.

1.2 Contributions

The main contribution of this thesis is a holistic, High Performance Computing approach to the hardware acceleration of genomics and sequence comparison, featuring a broad range of architectures and methodologies. Thus, each algorithmic proposal is fitted to a specific model of the computer architectural landscape and each research finding can be generalized to other disciplines, such as generic string-matching, which is a fundamental and core problem of computer science.

The presented work in this thesis comprises the following research items:

1. An optimized algorithm for the ultra-fast matching of short DNA strings for the comparison of massive genome data. In this work, the foundations for the efficient extraction of the signals and messages within DNA sequences are laid out, in order to enable previously unfeasible experiments in low-end, regular desktop CPU computers.
2. A parallel application that exploits shared-memory architectures for near-optimal speedup in fine-grained metagenomic pairwise comparisons. In this work, a parallel mechanism along with algorithmic optimizations are designed to cope with the huge complexity of optimal sequence matching in partially incomplete and noisy inputs.
3. An integral parallelization design and implementation on Graphic Processing Units of a state-of-the-art algorithm for the comparison of DNA sequences. This research proposes new fine-grained mechanisms to transform the heterogeneous granularity and unbalanced load of DNA comparisons into regular and non-divergent blocks of work suitable for these massively parallel devices.
4. As additional work, a machine learning approach for the prediction of resource consumption (in terms of time and memory) in DNA sequence comparison is presented. This work is aimed at enabling smart scheduling and dispatching of comparative genomics applications in supercomputers and cloud computing systems.

Contributions 1, 2 and 3 are accompanied by their respective publication in peer-reviewed journals. The last contribution was presented at a national conference.

1.3 Outline of this thesis

The structure of this thesis is similar to that of traditional research manuscripts. The existing cohesion between articles allows to present works in a successive fashion, linked and ordered by their relation in regards to the computer hardware architecture they make use of. Therefore, the Methods and Results are presented per each contributing work. This includes:

1. **Introduction.** The context of High Performance Computing and its application to research is briefly described.
2. **Background.** The computer architectures, optimization and parallelism techniques and algorithms for string matching are reviewed.
3. **Ultra-fast sequence comparison on single-processor systems.** The methodology and results for the first contribution (1).

4. **Shared-memory multiprocessing for the fine-grained alignment of reads.** The methodology and results for the second contribution (2).
5. **GPU-acceleration for the irregular and exhaustive sequence comparison.** The methodology and results for the third contribution (3).
6. **Efficient scheduling of DNA comparisons.** The methodology and results for the fourth contribution (4).
7. **Conclusions.** The outcome of this thesis is summarized and the future work is addressed.

Along with the main content of the thesis, other appendices are included to extend on further information that was omitted due to size and simplicity reasons:

- Appendix A "Hardware differences for exploiting performance in CPUs, GPUs and FPGAs".
- Appendix B "Sequential computation of k -mers".
- Appendix C "Custom memory pool allocator in ModernGPU merge sort".
- Appendix D "List of publications supporting this thesis".
- Appendix E "Resumen en Español - Summary in Spanish".

Chapter 2

Background

Research has become increasingly computational, especially since the advent of disciplines concerning Big Data, Machine Learning, particle simulation, molecular docking, *etc.*, which have rocketed the processing requirements of modern science. This scenario is promoting the acquisition and deployment of more and more computing infrastructure, resulting in the recent record-breaking of reaching over one exaflop (on mixed precision) for the first time in the history of supercomputing [1].

However, the implementation and deployment of these HPC architectures is not trivial. Traditionally, it required the installation of large and expensive computing infrastructure along with cooling equipment, massive power supply, additional protective mechanisms against hazardous elements such as dust or fire, *etc.* Besides these direct costs, it also required daily hardware maintenance (*e.g.* replacing faulty components), software supervision (*e.g.* manually supervising jobs or restoring users access), network security against intruders, *etc.* All these tasks required employing more personnel which also incurred additional costs, especially in the long-term budget. Other not-so-apparent concerns appeared which were more related to social and environmental issues, such as power consumption, impact on climate change, open and private user access, *etc.*

All in all, these reasons motivated huge interest towards cloud computing as a new way of providing computational resources with much higher flexibility for the final users. In fact, major stakeholders such as Amazon, Microsoft, Google or Oracle took notice and started to build large computing clusters, beginning the booming [2] of cloud computing services of the last decade. Ever since, numerous scientific studies requiring the use of HPC have been conducted on cloud infrastructure, such as distributed neural networks [3], remote sensing management [4] or weather prediction [5], among others. Still, even after all the immediate benefits that cloud computing has to offer to HPC research, such as short-term

cost savings, quick deployment, leveraged maintenance, *etc.*, several studies suggest that traditional supercomputing centers still offer higher performance at a similar cost (or even less) in the long run [6], [7].

It is without risk to say that the decision-making of choosing a platform for scientific computing is not a simple one, and things are further complicated as hardware accelerators are factored in. Over the last years, these massively parallel devices, particularly GPUs and FPGAs, have become intrinsically cheaper while offering comparable performance to that of tenths or even hundreds of CPU cores. Furthermore, these devices can be used on their own as small supercomputers, or they can also be included in supercomputing centers or cloud computing providers, therefore allowing researchers to carefully reason and select the optimal HPC approach to their specific research goals while balancing budget.

This scenario has propitiated a wider adoption of hardware accelerators within scientific research, up to the point where terms such as "GPGPU" or "General-Purpose Graphics Processing Unit" [8] have been coined to summarize the new use of these devices. Moreover, the increase computational capabilities of these devices have therefore helped to bridge the gap between the processing requirements and the current data-rich era. Fields such as machine learning, particularly in the training stages of neural networks, are nowadays unfeasible without the assistance of GPUs, FPGAs or even ASICs, such as the Tensor Processing Units (TPUs). While several other successful applications of general-purpose computing are recently being made public (*e.g.* electromagnetics [9]), there are still many fields and algorithms where further research on these devices is required, and bioinformatics is such an example. In essence, bioinformatics is defined as an interdisciplinary field that makes use of software to process and understand biological data [10]. While composed of many other subdomains (such as comparative genomics or sequence assembly), a core-problem which is recurrently predominant within bioinformatics is the comparison of biological sequences. Sequence comparison plays a major role since it enables to assess similarity between sequences and therefore identify functionality [11] and structure [12], understand evolution [13], *etc.*

Despite its regular presence in everyday Life-sciences analysis workflows, sequence comparison is still a computationally expensive step that has not been figured out completely. Therefore, understanding how sequence comparison works and improving current methods is key to developing even more efficient workflows that are central to so many other disciplines which in the end impact human health at a worldwide scale.

In the next sections, and in order to enable fluent and posterior discussion of the results, some of the most important elements of the computer architectures employed in this thesis will be introduced. Similarly, a background section concerning sequence comparison algorithms, data structures and their relation to well-known computer science problems is also included to give a complete perspective of the application domain.

2.1 Central Processing Units, Shared-memory Machines and Graphic Processing Units

According to Flynn's taxonomy [14], which is often used to describe the computer architecture hierarchy, most computer architectures are based on the four following types:

- **SISD (Single Instruction, Single Data):** one processing unit executes a single instruction stream over a single memory data. This corresponds to the traditional Von Neumann computer organization [15].
- **SIMD (Single Instruction, Multiple Data):** multiple processing elements that can execute a single instruction over multiple data, thus enabling data parallelism (but no task parallelism). Common CPUs include this type of vectorized instructions.
- **MIMD (Multiple Instructions, Multiple Data):** multiple processors executing instructions asynchronously and independently on multiple datasets. This category corresponds to totally parallel machines, such as distributed or shared memory supercomputers.
- **MISD (Multiple Instructions, Single Data):** multiple processors executing a different sequence of instructions over the same data. This corresponds to redundancy and fault tolerance systems, or devices where different parts of the hardware compute partial results on the same data, such as in systolic arrays [16].

Other extended types also exist (see [17]). It is noteworthy that with the exception of very rudimentary machines, most computers are a mix of the different types of architectures. On one hand, common processors are based on the SISD model, however they typically include several cores that can execute other processes over different data (thus similar to a MIMD model). Moreover, they also include vectorized instruction sets, thus resembling a SIMD. In this line, this manuscript will focus on single processors (SISD), shared-memory machines (MIMD) and GPUs (SIMD). In the following sections, their differences will be highlighted.

2.2 Acceleration potential by hardware architecture

CPUs excel at sequential execution, and they are prepared to be able to deal with highly heterogeneous work, making them specially suitable for every-day, irregular computing. Their design is focused on being able to run few sequential threads at the same time but each of them benefiting from advanced hardware optimizations: a high clock (in the order of 2 to 5 Ghz), multiple pipelines for higher output of instructions per cycle (IPC), branch prediction mechanisms to improve the flow in the instruction pipeline, out of order execution to increase IPC count by avoiding wasted cycles, extended instruction sets to perform vectorized operations, *etc.* Therefore, single-core applications usually benefit from adequate baseline performance, nevertheless with the limitation of being able to scale only by a few cores. In this line, shared and distributed memory machines deal with the problems of scaling regular processors.

In particular, shared-memory machines can be seen as a powerful (but limited) aggregation of CPUs which -when used well- will scale applications efficiently. Since memory is shared, their parallel programming is as transparent as it is with regular, single CPUs. Nevertheless, the potential parallel exploitation is limited precisely due to the abstraction on the memory interface, which is almost completely governed by the actual hardware implementation. This means that performance is subject to memory access patterns (*e.g.* multiple cores writing to the same location causes performance decrease due to contention). Another common drawback is the need of additional protocols to maintain the coherence of cache data. Further memory models, such as NUMA (Non-Uniform Memory Access) [18], where every multiprocessor has a local memory space, access patterns become increasingly more important as the latency of accessing non-local memory becomes several orders of magnitude higher. However, this in turn allows higher efficiency when scaling the number of processors, at the expense of a less transparent programming model. In a similar fashion, distributed memory machines differ themselves from shared memory machines in that there is no mapping of memory addresses between processors, thus requiring explicit synchronization mechanisms (such as Message Passing Interface, or MPI [19]) to access data located within the memory of another processor. The latency of these synchronization calls is usually orders of magnitude higher and therefore its handling becomes challenging, however at the benefit of huge scalability since the drawbacks of shared memory machines are not present.

GPUs on the contrary excel at highly-regular parallel workloads where the same set of operations must be applied to large volumes of data. Their design is focused on using many but simpler cores (compared to a traditional CPU), and their massive parallel processing power emanates from performing homogeneous computations (same work, same length, same locality, *etc.*) coupled with enormous memory bandwidth that serve as many cores as

possible at once. In this line, GPUs can be seen as a SIMD device (or rather a "Same Program, Multiple Data") which has even further restrictions when compared to a shared or distributed memory system. Besides their memory bandwidth being completely dependent on pattern access (*i.e.* coalescence), all cores within a warp (or group) must work together in lockstep. This means that branching (*e.g.* conditionals) must be avoided unless all working cores are taking the same execution path or otherwise the performance is decreased strongly due to serialization. These, along with other drawbacks and techniques, cause the programmer to be much more restricted when exploiting the massive parallelism capabilities of these devices.

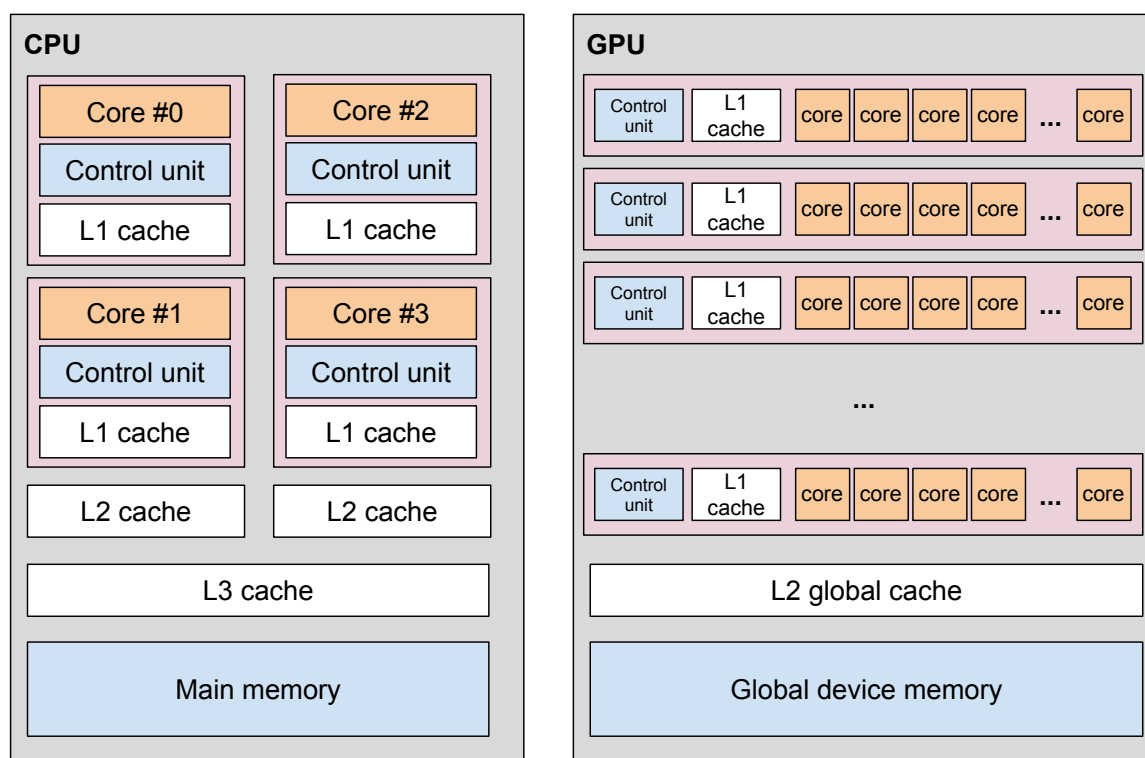


Fig. 2.1 Coarse-grain comparison of the layout of components within a CPU and a GPU. (Left) A quad-core CPU is shown featuring four cores divided into two blocks, each sharing an L2 cache and an L3 cache available globally. (Right) A GPU showing the cores or Streaming Multiprocessor (in NVIDIA architecture) as a row, each containing the pertaining control unit, scheduler, *etc.*, and an L1 cache. An L2 global cache is shared similarly to the L3 in the case of the CPU.

It is noteworthy to include a short note regarding Field Programmable Gate Arrays (FPGAs). While CPUs and GPUs represent a trade-off between job regularity and parallel performance, FPGAs trade flexibility and specificity for incomparable speed. This is, FPGAs enable to design the logic in the circuit to best suit an algorithm (as if designing the instruction set of a processor); however at the expense of much less versatility in the programming,

less deployment flexibility and a much more steep learning curve. Since the logic can be tailored to the application, it is possible to achieve higher performance and also high levels of parallelism, which are mostly limited by the die size and propagation delay, among others. The architecture of an FPGA typically includes an array of programmable logic blocks and routing channels (so that blocks can be wired together to perform different functions). Memory blocks are also commonly included to avoid the wasting of logical blocks for memory purposes.

Further discussion regarding the hardware differences between CPUs, GPUs and FPGAs is available in Appendix A "Hardware differences for exploiting performance in CPUs, GPUs and FPGAs".

2.3 Sequence comparison

Sequence comparison or sequence alignment is the attempt of determining the best way in which letters from two different sequences can be arranged so that the number of changes required to convert one into the other is minimized. From a mathematical point of view, the sequence alignment problem can be seen as an optimization problem in which the objective is to minimize the number of differences between two sequences according to a set of penalties and/or rewards imposed by the optimization function.

Thus, if two sequences are similar, then there will be a small number of differences between the two. Likewise, if the sequences are largely dissimilar, lots of differences will arise. From a biological point of view, the differences between two sequences (under the premise of a common origin [20]) can be interpreted as mutations and rearrangements, meaning that the two sequences or species shared a common ancestor from which they both diverged slowly over time. However, due to errors in replication and the impact of external factors (such as radiation or other agents) [21], the sequences started to diverge. A similar process occurs with written text over time, where *e.g.* many words remain the same (no mutations), others have changed just slightly (single point mutations, see [22]), other words have disappeared completely (deletions [23]) and lastly others would be new (insertions [24]). Additionally, more complicated events could have also taken place, such as mixing, reversing or reallocating, but discussing these falls out of the scope of this manuscript.

In the next sections, the following will be introduced: (1) mathematical definitions of DNA sequences, (2) their comparison to common string matching techniques and (3) a review of the most common algorithms for sequence alignment.

2.3.1 Definitions

DNA sequences are composed of nucleotides. These are computationally represented with letters. Consider the finite alphabet $V = \{A, C, G, T\}$ which comprises the four existing nucleotides [25]. Notice that other letters exist in database sequences to account for possible errors in the sequencing process (*i.e.* the “reading” of the DNA sequence and transformation into text) but these can be overlooked, particularly when considering highly-curated information such as that of genome sequences. To remove these unwanted characters, a pre-processing over the input data is usually performed.

A word w is defined as any finite string over the alphabet V . Thus, a word w built over the alphabet V must be accepted by the regular expression $[ACGT]^+$. The set of all possible words w over V is denoted as the universe U . Now, a sequence s is a word of size n such that the length of s is n , *i.e.* $|s| = n$. Consider all the words of size k that are contained in sequence s as the set $C = \{w_1, w_2, \dots, w_m\}$, where $m = n - k + 1$ and each sub index corresponds to the position of the word in the sequence. The set C contains all possible overlapping words of the sequence s with fixed size k , such that $|w_i| = k$. Given two sequences s_a and s_b , the set of seeds (also known as “hits”) between both is:

$$H_{a,b} \subseteq C_a \times C_b \mid \forall h = (w_i, w_j), w_i \in C_a, w_j \in C_b, w_i = w_j \quad (2.1)$$

The above definition corresponds to the set of exactly equal words $H_{a,b}$ between sequences s_a and s_b , which is a subset of the cartesian product $C_a \times C_b$ between the collection of overlapping words of each sequence. So far, the definitions include:

- A sequence as a string of letters that can only contain A, C, G, T , *e.g.* ACCGTTGA.
- A word as a subsequence of letters of fixed size k , *e.g.* for $k = 3$ the word w_2 in the previous example is CCG.
- The set of overlapping words of fixed size of a sequence C .
- The set of matching words between two sets C_a and C_b of overlapping words of fixed size, $H_{a,b}$.

At this point it becomes interesting to note that the parameter k has an enormous effect on the distribution $H_{a,b}$. That is, for long enough sequences, *i.e.* $n \gg m$ and assuming a uniform distribution for the probabilities of each letter, a small k means that the distribution $H_{a,b}$ contains nearly all tuples $(w_i, w_j) \mid w_i \in C_a, w_j \in C_b, w_i = w_j$. On the contrary, a large

k reduces the number of elements in the set $H_{a,b}$. In essence, this means that the larger the words being looked for, the harder it is to find them (assuming any with non zero probabilities). This can be explained as follows: since each letter has a non-zero probability of appearing, finding the exact combination of letters $x_1x_2x_3x_i...x_k$ becomes geometrically harder since the random probability of matching two such words is $\prod_{j=1}^k p(x_j)$. Intuitively, the larger the word, the harder it is that all the letters are equal under a randomly generative process.

2.3.2 Differences with common string matching

The problem of sequence comparison is similar to that of generic text matching or pattern matching. However, both matching algorithms have been adapted over time to improve performance based on the specific characteristics of the application domain, however at the expense of reducing their versatility and generalization. In this section the differences that make each field unique will be discussed.

Difference in Alphabets

The first difference is the alphabet. In the case of DNA, the reduced alphabet $V = \{A, C, G, T\}$ is utilized as opposed to that of a complete language. The alphabet V sometimes needs to include a fifth letter (see the IUPAC code [26]) to account for unknown nucleotides, resulting in the extended alphabet $V' = \{A, C, G, T, N\}$. Both V and V' are clearly a subset of the common alphabet for text matching, which includes from all letters and numbers, to symbols and even worldwide character encodings such as UNICODE [27]. In any case, a reduced alphabet such as V has enormous computational advantages over traditional alphabets. For instance, when indexing hash tables based on strings, a reduced alphabet enables to find much faster hash functions. In fact, in the case of DNA, a perfect (free of collisions) indexing hash function f for the universe U^k (*i.e.* all existing words w of size k over V) is given by the following function:

$$f(w_i^k) = |V|^0 \times v(w_{i,1}^k) + |V|^1 \times v(w_{i,2}^k) + \dots + |V|^{j-1} \times v(w_{i,j}^k) + \dots + |V|^{k-1} \times v(w_{i,k}^k) \quad (2.2)$$

$$f(w_i^k) = \sum_{j=1}^k |V|^{j-1} \times v(w_{i,j}^k) \quad (2.3)$$

Where $v(x)$ simply returns a unique numeric value for each letter of the alphabet V (e.g. $v(a) = 0$, $v(c) = 1$ and $v(g) = 2$, etc.), and $w_{i,j}^k$ denotes the j -th letter of the word w_i with $|w_i| = k$. It follows from Equation 2.3 that function f is a bijection $f : U^k \rightarrow N^{1..|V|^k}$ that creates a perfect indexing with no collisions for U^k . Moreover, f is a monotonically increasing function with equal number of images in the codomain and domain, therefore accounting for 4^k possible values for the alphabet $|V| = 4$. Notice however that for large k values the size of the function domain grows exponentially, and thus a relatively small $k = 16$ yields a domain of 4,294,967,296 elements. If the memory required for each element is factored in (such as a long integer requiring 8 bytes), the required memory increases up to approximately 35 Gigabytes of RAM, which is unsuitable already for most desktop computers. It can be seen how this approach quickly runs out of space.

Regarding common string-matching hash functions, it is not as easy to find such simple functions with near free-collisions, and it typically requires additional and expensive preprocessing of the input sequences to find the best suitable function [28].

Types of patterns

Another difference that arises between DNA string matching and text matching is the fact that the latter case is normally focused on finding the instances of one single pattern across a larger text. In the case of DNA, the extreme opposite takes place: algorithms need to find not only multiple patterns, but all possible patterns. There are two consequences to this, namely:

- Since only one or few patterns need to be searched for, the required resources are automatically smaller. Instead of requiring a complex data structure capable of handling N patterns for an $N \times M$ pattern comparison, the only requirement is to save one pattern and match it against the text to be searched split by batches, hence reducing memory overhead.
- If the pattern to be searched comes from a static collection of patterns, a common technique is to *a priori* build a deterministic finite automata that accepts the pattern, and then scan the reference text through the automata. In fact, a Deterministic Finite Automata or DFA is only bounded by the input corpus text that is search for, therefore requiring only $O(n)$ time [29] (where n is the number of elements in the input text). However, these automatas are often limited to small and static pattern sizes, or otherwise they might require an exponential number of states (and construction time), which translates in exponential space. Although conversion to Non-deterministic Finite Automatas (NFA) is possible, these take quadratic run time $O(m^2n)$ (where m is

number of nodes as constructed from the pattern) since every possible path must be explored.

In the case of DNA sequence matching, it is not possible to take the above mentioned simplifications, for all patterns must be searched for. Of course, it would be possible to preprocess one of the input sequences to generate all of the required automatons (one per different word w) and then feed all of the other sequences in parallel to all of the models. However this would be less than efficient.

Still, interesting initiatives have been developed in recent years to overcome the limitation of a small number of patterns, especially since the availability of cheaper hardware accelerators such as GPUs or FPGAs. One of the most widely used methods works by developing a compiler for regular expressions that translates these into an instruction set for a specific, hardware deployed processor [30]. The patterns translated into code instructions (such as jumps or boolean logic) are then fed to the hardware accelerator that runs the processor along with batch splits of the text to be searched. The result is that the specific processor matches the patterns by executing the program containing the coded patterns. A recent example of this approach developed for FPGAs is available in [31]. Interestingly, one of the benefits of translating the regular expressions to instructions is that many techniques from modern computer architecture can be applied, such as branch prediction [32] or cache prefetching [33].

Number of patterns

Another challenging and recurrent problem in DNA matching is dealing with highly repetitive DNA regions. In the context of regular string-matching, this could be seen as the problem of dealing with small articles or pronouns that do not really add much to the meaning of a sentence, but rather serve a structural purpose. However, the same reasoning does not hold in the case of DNA sequence comparisons, and therefore all these repetitive words cannot be discarded. Even worse, while in common books or texts the number of words might be manageable, the sheer size of DNA sequences makes the handling of these repetitive regions extremely difficult since their multiplicative product (*i.e.* a single word from one sequence is matched to all of the other sequence) becomes unfeasible even in “smaller” sequences. Figure 2.2 shows an example of the huge number of matches between two sequences. At first glance, it seems that the distribution of matches is governed by two distributions, one relatively uniform and one –apparently– exponential. Note that the latter one is actually of quadratic form in the worst case (*i.e.* matching a collection of n equal words to the collection itself, thus resulting in n^2 matches), and that only the x-axis needs to be adjusted in terms of

the number of words rather than the number of linear increments of DNA sections. In fact Figure 2.3 shows the right tail of the comparison along with its quadratic fit.

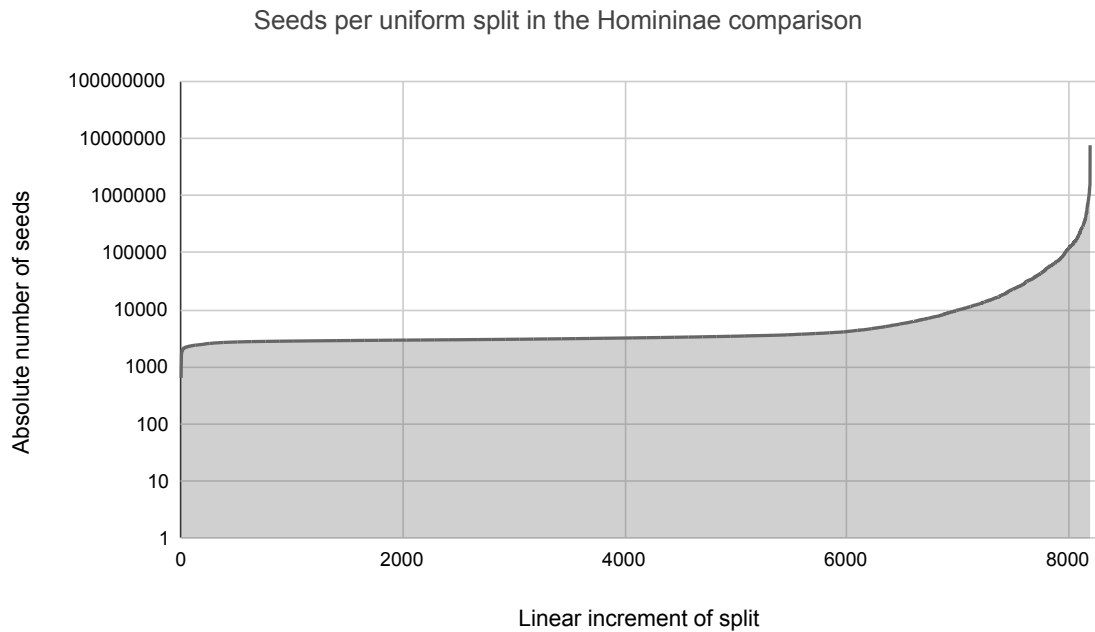


Fig. 2.2 Number of seeds generated per equally-sized DNA sections. The x-axis represents linear increments (of approximately 4,000 bp) of an arbitrary DNA section. The $\log_{10}$ y-axis shows the absolute number of seeds found per each of these sections.

Length of patterns

Another interesting aspect is the difference between length in the pattern to be searched. On one hand, in regular string matching the patterns that are looked after are typically around a few words or sentence size. The performance of current models quickly degrades with longer sequences, since they are not designed for such purpose [34].

On the contrary, in DNA the size of the patterns is not only unpredictable, but also variable by several orders of magnitude. Figure 2.4 illustrates such behaviour by showing the length of the patterns matched between the two X chromosomes of the *Homo sapiens* and *Mus musculus* species. Notice that (1) the number of patterns matched ascended to $n = 15,199,400$ and (2) the variability in length between the smaller and larger patterns is of almost five orders of magnitude.

Such variability makes the use of traditional string-matching algorithms unfeasible for the general case of sequence comparison. In the following sections, a review of the most common

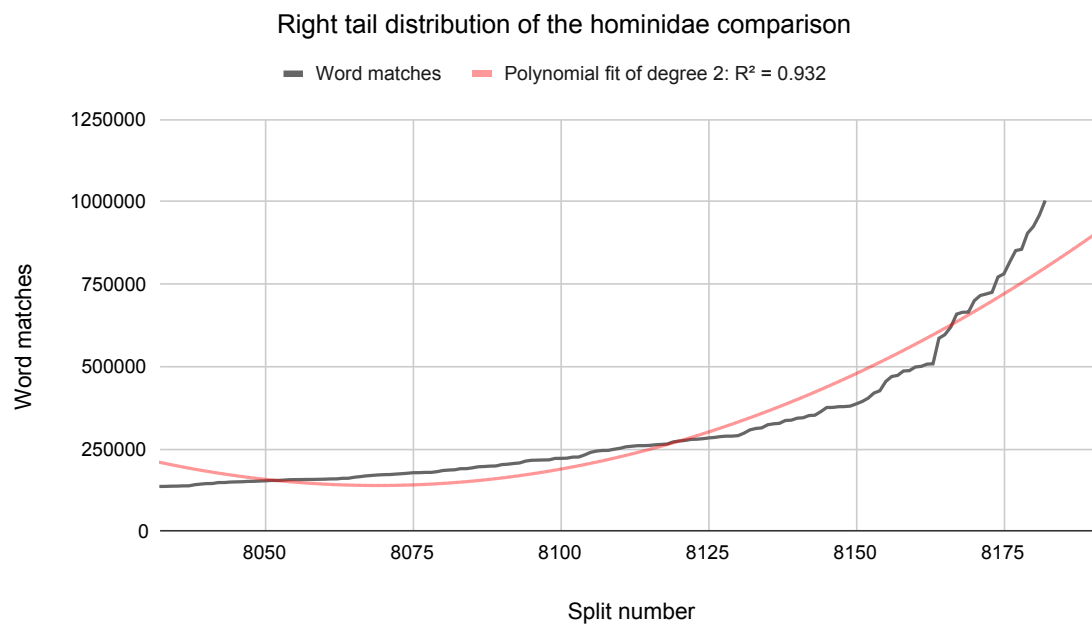


Fig. 2.3 Right tail distribution of the Hominidae comparison (Figure 2.2). The x-axis represents the last 150 linear increments of approximately 4,000 by 4,000 bp each.

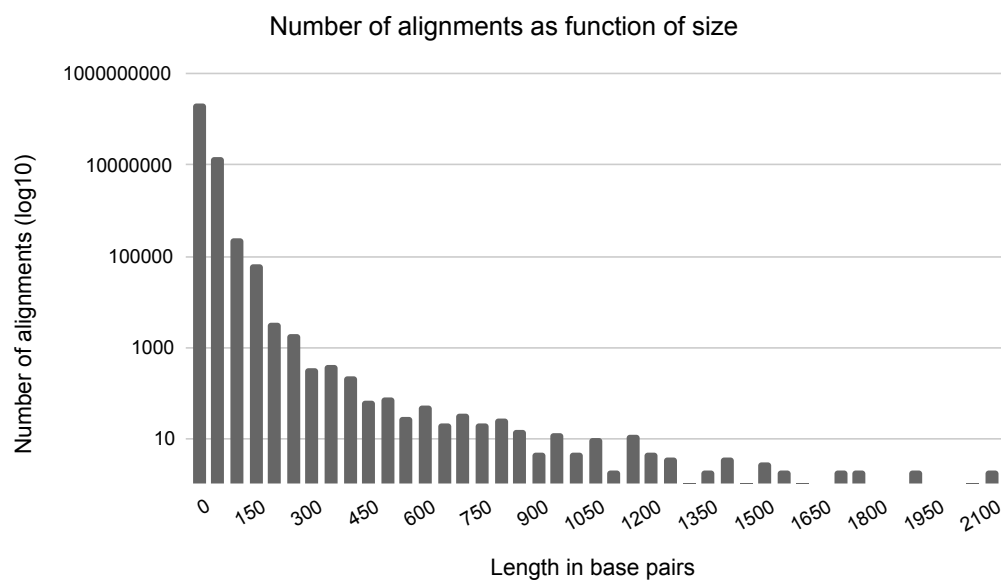


Fig. 2.4 Distribution of the length of High-scoring Segment Pairs (HSPs, or the “patterns” found) in the comparison between *Homo sapiens* chr. X and *Mus musculus* chr. X. The y-axis is in $\log_{10}$ scale.

algorithms developed in the last decades to perform sequence comparison is presented along with an analysis of their complexity.

2.3.3 Algorithms

In this section, the most common approaches to sequence comparison are introduced. Given the differences and similarities between generic pattern matching and DNA matching discussed previously, many of the following algorithms are applicable for either DNA or general string matching if certain constraints are satisfied. However, some algorithms might not be as suitable depending on the lengths of the alphabet V . Sequence comparison methods can be divided into roughly three groups depending on the precision achieved and the computation required: (1) alignment-free methods, (2) dynamic-programming alignment methods and (3) seed-based aligning methods. Each of these methods serves a different purpose and at the same time, each has its advantages and disadvantages in terms of the optimality of the matched patterns and the algorithmic complexity. In the following subsections, each methodology will be addressed individually.

Alignment-free methods

The group of alignment-free methods includes any algorithm that does not require alignments nor partial pattern matching. Such a permissive condition puts in the same bag a wide variety of algorithms.

Dotplots The simplest one is the dotplot technique [35], which is computed as a double loop where each letter in both sequences is compared individually. The obvious problem with this algorithm is the quadratic complexity in both time and space. This algorithm is suitable for very small sequences and serves principally as a visualization technique. Figure 2.5 represents a sequence comparison performed with a dotplot method.

Notice from Figure 2.5 that the left comparison includes a dark diagonal accumulation from top-left to bottom-right. This corresponds to the signal between closely related sequences. A dotplot has basically three geometrical features used to indicate the degree of similarity: (1) diagonals, (2) horizontal and vertical lines and (3) single points. The first one corresponds to contiguous signals in both sequences, *i.e.* they are nearly identical from start to end of the diagonal. The second one corresponds to repeats from one sequence into the other. Square areas represent accumulation of repeats. The single points indicate random matches between nucleotides.

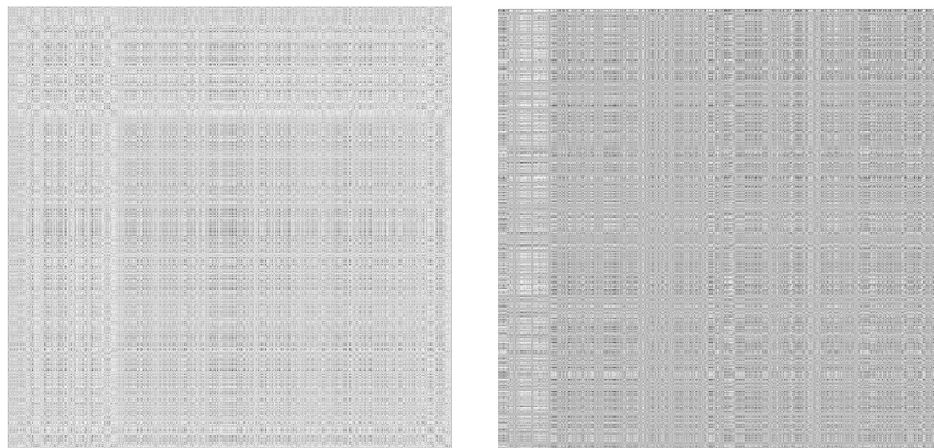


Fig. 2.5 Two dotplot comparisons. (Left) Comparison of *Homo sapiens* coagulation factor XII (F12). (Right) Comparison between the SARS-COV2 surface glycoprotein and mitochondrial membrane 70. A darker color indicates a higher number of matched nucleotides.

On the computational note, dotplots can easily be improved by matching whole words instead of single nucleotides. However, sensitivity decays as the size of the word k increases due to single nucleotides being present but larger words not, especially considering that even single mutations would result in whole words being discarded.

K-mer based approaches These approaches are not specifically designed for sequence comparison in a traditional sense, but rather to extract a series of features on which sequences can be classified, studied and compared. However, although sequence comparison based solely on the study of k -mer distributions can be extremely fast, it is worth noting that it does not come with a high level of precision nor accuracy, and therefore these methods are mostly used for classification and clustering of large sets of DNA sequences [36].

In [37], researchers used the k -mer spectra of over one hundred genome sequences to characterize their distribution modality, which was utilized for evolutionary studies under the assumption that a similar modality implies a certain relatedness between sequences. Still, such characterization can be further used for clustering sequences together as discussed in the next example.

A classical example of k -mer models for sequence DNA comparison is the use of Markov models [38]. These models can be applied to DNA sequences by calculating the transition matrix which effectively translates to the transition frequency of k -words. As an example, in [39] the authors combined Markov chains with k -mer distributions to design new statistical measures that could be used to discriminate between sequences. This is particularly useful in the event of performing a database search with a given query sequence, as it acts as

a sequence discriminator that enables to quickly prune the search space and thus reduce runtimes.

Finally, a different method used to compare sequences (even if it is not strictly based on k -mers but on single nucleotides, and they will be considered within the case of $k = 1$) makes use of the sum of joint probabilities per nucleotide with its predecessors [40]. This enables the authors to extract a descriptor of sixteen variables which can be further compared to the descriptor of other sequences by using the Euclidean distance.

Dynamic programming methods

Dynamic programming (DP) [41] is an optimization technique. In essence, the algorithm recursively divides a problem into smaller subproblems, such that if the smaller subproblems can be solved optimally, then the larger problem can be solved optimally as well by making use of the aggregated, individually-optimal solutions. These two properties of a problem are known as optimal substructure and overlapping-subproblems.

The relationship between DP and sequence comparison comes from the fact that the problem can be seen as finding the minimum number of changes required to convert one sequence into the other. Sequences Q (query) and R (reference) are split naturally by nucleotides, which represent the different time steps of the problem (and thus there are $n * m$ decisions, accounting for the product of the lengths of Q and R). At each of these timesteps, a decision must be taken which yields a value $v(i, j)$. Three decisions are normally modelled, either inserting, deleting or matching the nucleotides in the given timestep. Thus, the recursion function to transform a query into a reference at state $(Q_{i,n}, R_{j,m})$ is defined as:

- Inserting the first nucleotide of the reference and performing the rest of the alignment with the current query and the tail of the reference $(Q_{i,n}, R_{j+1,m})$. This corresponds to a gap in the reference.
- Deleting the first nucleotide of the query and performing the rest of the alignment with the tail of the query and the current reference $(Q_{i+1,n}, R_{j,m})$. This corresponds to a gap in the query.
- Matching the first nucleotide of both query and reference sequences and perform the rest of the alignment with the resulting tails $(Q_{i+1,n}, R_{j+1,m})$.

The particularities of the recursion are specific to each of the DP approaches to sequence comparison. These can be represented by two methods, namely (1) the Needleman-Wunsch algorithm [42] and (2) the Smith-Waterman algorithm [43]. The difference between the two is their ultimate purpose. On one hand, the Needleman-Wunsch algorithm (NW) was designed

to perform global alignments, *i.e.* fully align (or “convert”) one sequence into the other, from start to end. On the other hand, the Smith-Waterman algorithm (SW) was designed to perform local alignments, which focuses on detecting all subsequence alignments as opposed to fully aligning one sequence to the other. As a result, the two algorithms are applied in different scenarios, in particular the NW algorithm is used mostly to compare closely-related sequences with fine-grained detail whereas the SW algorithm is used to compare sequences that are less related and feature more evolutionary events, mutations, *etc.* This makes the SW algorithm also more suitable for the comparison of other types of sequences besides genomes, such as metagenomes [44].

These differences can be appreciated through the DP recursion functions for each of the algorithms, see below (NW is shown first in Equation 2.4, followed by SW in 2.5).

$$\begin{aligned}
 F_{0j} &= d * j \\
 F_{i0} &= d * i \\
 F_{ij} &= \max \begin{cases} F_{i,j-1} + d \\ F_{i-1,j} + d \\ F_{i-1,j-1} + S(A_i, B_j) \end{cases} \quad (2.4)
 \end{aligned}$$

$$\begin{aligned}
 H_{i0} &= H_{0j} = 0 \\
 H_{ij} &= \max \begin{cases} H_{i-1,j-1} + S(A_i, B_j) \\ \max_{k \geq 1} (H_{i-k,j} - d * k) \\ \max_{l \geq 1} (H_{i,j-l} - d * l) \\ 0 \end{cases} \quad (2.5)
 \end{aligned}$$

Note the following algorithmic differences: On one hand, the SW algorithm enables cells to have a value of 0, which implies either no alignment or the end of one. This is to allow discontinuous (or local) alignments. In the final backtracking stage, the highest values in the scoring matrix can be used to trace alignments back until a value of 0 is reached. On the other hand, the NW recursion allows negative values, which serve to model low similarity. The final backtracking stage has therefore to begin from the ending cell $v(n, m)$ until it reaches the start of the alignment which must be at $v(i, 0)$ or $v(0, j)$.

Still, these two algorithms require quadratic time and space. The time restriction comes from having to take a decision at every $v(i, j)$ timestep, whereas the space restriction comes

from the requirements of the backtracking stage itself, since every value $v(i, j)$ must be memoized (as part of the DP approach to reuse suboptimal structures) in order to enable the reconstruction of alignments. Therefore, the complexity of the NW or SW algorithms is (without any optimization) $O(n * m)$ in both time and space. This is feasible for small sequences (*e.g.* some genes) but not for massive genomes. Thus, a partial solution that mitigates the complexity was designed by [45] which introduced a divide-and-conquer approach to reduce the complexity from quadratic to linear in terms of space. The key idea behind the algorithm is that a midpoint q must exist in the alignment such that it partitions the sequences in two. Luckily enough, the midpoint q can be calculated using only $O(n + m)$ space (the length of the longest sequence plus the maximum number of recursive calls saving one cell each) since the memory requirement in the DP approach can be overridden if no memoization is to be done. Therefore, successively finding these midpoints results in a divide and conquer approach where every partition makes the search smaller until the leaves of the recursion tree (which correspond to the indices of the midpoints) contain the decision taken at every timestep of the alignment.

Seed-based and heuristic aligning methods

In the previous section, algorithms for the optimal alignment of DNA sequences were discussed, particularly in terms of their complexity, which are quadratic in time and space (or linear if using Hirschberg's method). The biggest drawback of the optimal aligning methods is clearly their enormous search space. However, exploring the whole of the search space is not always necessary. Therefore, seed-based methods will be introduced now, which are designed to heuristically prune the search space by first looking for “seeds” (or fixed word matches) that will be extended afterwards. The reasoning behind this approach is simple yet extremely effective: a good alignment is likely to have a consecutive number of exactly aligning residues (or matching nucleotides). Such a statement will stay true for most realistic genome sequence comparison scenarios, since it has been shown that DNA regions that have an important role in the living of an organism are less prone to evolutionary changes over time [46] in an attempt to stay alive.

The procedure for any seed-based aligning method includes roughly three steps, namely (1) indexing of words, (2) matching of similar words into seeds and (3) extension of seeds into alignments. Notice that the detection of the seeds requires prior identification of similar words. If performed naively, as in the dotplot approach, such an algorithm would require quadratic time. Nonetheless, in the next sections the algorithms and data structures to efficiently perform such detection and matching will be described along with the method for the extension of seeds into alignments.

Seed optimization The purpose of indexing words is to reduce the time complexity (and memory in some cases) of querying whether a word is present in both sequences being compared. Once such a data structure exists (see next sections), the next step in the algorithm is to extend the matching words (seeds) into larger alignments. These are known either as High-scoring Segment Pairs (or HSPs) if they present no gaps (*i.e.* a continuous stream of nucleotides), and “gapped alignments” if they do present gaps.

The extension can be done roughly in two ways depending if the resulting type of alignment is an HSP or a gapped alignment. In the first case, since no gaps can be inserted into the alignment, the algorithm is a linear optimization that runs both forward and backwards in respect to the initial seed. That is, a reward is assigned to matching similar nucleotides together and otherwise a penalty if they are unequal. Therefore, the optimization problem is to find the pair of offsets p_1, p_2 that maximize the alignment score $s_{i,j}$ for every seed:

$$s_{i,j} = \arg \max_{p_1, p_2} \sum_{l=-p_1}^{p_2} g(q_{i+l}, r_{j+l}) \quad (2.6)$$

In equation 2.6, i and j are the coordinates of the seed in the query and reference sequence and the function $g(q_i, r_j)$ returns 1 if $q_i = r_j$ and -1 otherwise. Note that the offsets p_1 and p_2 must remain within the boundaries of both sequences, *i.e.* $p_1 < \min(i, j)$ and $p_2 < \min(|q| - i, |r| - j)$.

Notice that the summations are practically from the central point of the seed up until infinity (or the end of the sequences, as described before). This means that a stopping condition must be set in order to prevent the algorithm from excessively trying to find the argument which maximizes the optimization. This condition is usually the convergence of the optimization towards zero, *i.e.* an initial score is assigned to the seed and the reward or penalty is added to it such that unequal nucleotides decrease the total sum whereas equal nucleotides increases it. This enables the optimization procedure to stop when the value reaches zero.

In the case of a gapped extension, usually optimal dynamic programming approaches are employed, although some methods introduce bounded searches to reduce the quadratic time and space to only a predefined strip of the whole search space.

Data structures

As discussed previously, seed-based methods are the preferred method for general-purpose sequence comparison due to their versatility. Therefore, it comes as no surprise that lots of research has been put into studying efficient algorithms and data structures to perform the

indexing and matching of words (which typically consumes most of the processing time). In this section, the most commonly used approaches are reviewed along with their advantages and disadvantages.

Hash tables Hash tables are one of the most widely used approaches for several reasons. For small word size values (around $k < 16$), a perfect collision-free hash function can be used to index words in $O(1)$ at the expense of a manageable space requirement of $O(4^k)$. Collision-free hash functions also have the additional benefit that the locations of words can be stored as chained linked lists to each slot of the table, thus enabling the querying of words in $O(1)$ as well, or even in fixed-size contiguous arrays to also benefit from cache locality and prefetching. However, it becomes apparent that for longer k word sizes the table will not fit into RAM, and modified hash functions need to be used that can no longer guarantee collision-free indexing, thus increasing both index and query times depending on the function used. Algorithms such as BLASTN [47] and CHROMEISTER [48] make use of these types of hash tables.

A hash-table based technique are bloom filters [49], which are probabilistic, space-efficient data structures that can be used to index words at an increased cost in time due to requiring the calculation of additional hash functions (*e.g.* three is common for DNA). Each of the hash functions sets a particular value of the hash table to one, and thus a word is contained if all of its hashed bits are set to one. This enables to save space since a single bit can be related to several words. Nevertheless, bloom filters are not specifically designed to store the values of elements but rather to model their belonging to a set and therefore additional overhead is needed to store the actual data such as the location of words.

Finite automata approaches Finite Automatas (or FAs) were used mostly in the early 90's by protein-search programs such as BLASTP [50]. However, their use was quickly abandoned in favor of hash table approaches, since a deterministic FA that runs in linear time requires an exponential number of states, whereas a non deterministic automata requiring linear number of states while running on quadratic time. This makes either the building of the automata or the accepting running time rather expensive and difficult to handle within memory. Variations to FAs, such as the Aho-Corasick algorithm for general purpose string pattern matching have also been applied to the specific case of DNA [51]. However, unlike other data structures (*e.g.* see trees below) the Aho-Corasick algorithm search time increases linearly with the number of patterns and thus in the general case of matching all words against all words (see section 2.3.2 "Number of patterns") the complexity becomes unfeasible.

Tree-based structures Binary trees are a simple solution as data structures for indexing and querying. However, raw binary trees require (1) hashing of words in order to speed up the process of comparing nodes, (2) extensive memory consumption to handle just pointers and (3) $O(\log n)$ steps to search a tree with $O(n)$ being the worst case if the tree becomes linear. Although the latter is not typical, near-linear substructures can be found in trees due to the unbalancing caused by massively repeated words.

Suffix trees are widely used for the indexing of words in sequence comparison algorithms. This is due to the two great benefits they offer, namely that they can be built in $O(n)$ time while allowing for subsequence search (thus enabling mismatches). However, as with binary trees, a lot of memory (typically $O(n \log n)$) is required to save all the pointers, which becomes highly impractical given the length of genomic sequences. Suffix arrays take only $O(n)$ space by removing the necessity to store all the pointers required for tree-structures. However, they require sorting the entire structure which is an additional $O(n \log n)$ and two binary searches to query an item. The small amount of space they require to be built make them appropriate for the indexing of large sequences. However, the fact that two binary searches are required (even if only $O(\log n)$ steps are needed per search) makes them not as efficient for querying many elements (just as in genome comparison). This is especially true when compared to hash table approaches which take $O(1)$ per query, which, in practical terms, implies that any method requiring binary search will take nearly $m * \log n$ more steps to compute than a hash table approach, being m the number of queries and n the size of the input string.

An example of a sequence comparison application that switched from suffix trees to suffix arrays is MUMMER [52].

Sort-based approaches Another simple but effective approach is sorting all words found in both sequences such that no querying of individual words is required afterwards, but rather a linear sweep of both word dictionaries. This approach requires $O(n \log n)$ for each sort and a single $O(n + m)$ for the matching of words. While the complexity in theory is higher, the actual matching of words is faster since no binary searches are needed. However, more memory is needed (and an additional sort) since both input sequences need to be preprocessed. Nevertheless, this approach has the benefit that the matching of words does not degrade as the sequences grow, contrary to what happens with binary search methods. An example sequence comparison algorithm that employs this algorithm is GECKO [53].

Data-compression methods The Burrows-Wheeler algorithm was originally developed for lossless data compression, but in conjunction with suffix arrays and the FM-index algorithm [54] it can be applied to indexing massive sequences. The key point of the method is its space

for time trade-off, meaning that whole genomes can be efficiently indexed while slightly requiring additional computation. This is achieved through the FM-index algorithm which enables to query a genome compressed of size $O(n)$ with the Burrows-Wheeler transform in $O(|m|)$ time, being m the size of the query string [55]. The building of the index takes $O(n \log n)$ time due to the required sort of the transformation. Moreover, the search finds all substring occurrences and thus makes the algorithm highly suitable for inexact or partial matching of small sequences against large genomes. Notice however that since the search requires $O(|m|)$ it loses performance as the size of m grows. An example software application is the short-read to reference mapper BOWTIE [56].

As a final remark, it is interesting to note that most of these algorithms share quite similar theoretical complexities, both in time and space. However, each approach benefits from different scenarios depending on the type of inputs (short versus long sequences), whether indices are required for both sequences or only one, the size of the words, *etc.*

2.4 Machine Learning

The field of Machine Learning includes methods that learn functions from sets of data (typically, past or historical data) in order to generalize to new, unseen data [57]. The most common form of learning, namely supervised learning [58], is based on supplying a set of labeled data, called training set, which includes input examples and their desired target output, and then minimizes a loss function (for example, using backpropagation [59]) in order to make better predictions on posterior data. The other two major branches of Machine Learning are unsupervised learning [60] and reinforcement learning [61]. The aim in these two approaches is widely different: unsupervised learning usually deals with extracting patterns or sets of rules from data (*e.g.* clustering), whereas reinforcement learning is more concerned with learning actively from the interaction with the environment through actions and rewards (*e.g.* games such as chess [62]).

In the following sections, three well-known methods for supervised learning will be briefly introduced, namely decision trees [63], gradient boosting [64] and neural networks [65]. These methods will be used and compared in the study regarding the automatic prediction of required computational resources in supercomputers for the problem of sequence alignment.

2.4.1 Decision trees

Decision trees are among the simplest techniques for supervised learning. They are modelled as a set of rules that recursively partition the input data, such that every node in the tree

splits the data according to the most optimal variable. The strategy to choose such variable is usually based on entropy or the Gini index [66]. While decision trees are mostly used for classification, they can also be used for regression. On one hand, some of the reasons decision trees are extremely popular are (1) simple interpretability, (2) no need for preprocessing and normalizing (as opposed to *e.g.* neural networks), (3) no assumptions are required in the input data, (4) ease of implementation, *etc.*

On the other hand, decision trees include numerous drawbacks that make them impractical in many scenarios, such as (1) not being very robust (small changes in the data result in large changes in the prediction tree), (2) they can easily overfit, (3) difficulty in capturing relationships between multiple variables, *etc.*

Besides their weaknesses, a powerful technique that builds on top of decision trees is called Random Forest [67], which makes use of an ensemble of decision trees to improve the accuracy of the predictions. The key idea behind this technique is that by employing multiple decision trees (created with randomization under a set of criteria [68]) and averaging the output predicted value, or selecting the top one, it is possible to overcome the limitations caused by their tendency to overfit the data, therefore greatly improving both accuracy and robustness. The biggest drawback to this process is the loss of interpretability, which in turn is one of the strengths of decision trees.

2.4.2 Gradient boosting trees

Similarly to decision trees, gradient boosting is mostly employed for classification and regression tasks. Gradient boosting is in itself a form of ensemble learning, *i.e.* building a set of models and aggregating their result to produce a better prediction [69]. However the main difference stems from the iterative nature of the algorithm, where instead of averaging individual estimators together (as in a Random Forest), each estimator attempts to minimize the error produced by the previous estimator. In essence, this is equivalent to having several weak predictors which can handle certain parts of the data but not its entirety, and therefore the focus of each estimator is directed towards a part of the data that is not explained by the remaining ones.

These weak estimators are typically decision trees, although other based models such as linear regression or radial basis functions can be used, among others [64]. In the same line as with Random Forest approaches, gradient boosting suffers from loss of interpretability. Another drawback when compared to random ensembles of decision trees is the fact that the training of gradient boosting ensembles are much harder to parallelize due to their iterative nature, as opposed with Random Forest where the decision trees can be built in parallel

without penalization. This makes the training stages of gradient boosting methods much longer, although resulting in a usually higher prediction performance.

2.4.3 Neural networks

Artificial neural networks are the preferred machine learning method for learning complex problems [70]. They have been proved to be exceptionally good at image processing tasks [71], time series and pattern recognition [72], natural language processing (especially since the advent of transformers) [73], anomaly detection [74], autonomous driving [75] and complex games [76], among many others.

Inspired by biology, artificial neural networks are composed of a large mesh of neurons connected together. Each connection carries an associated weight which is aggregated on a per-neuron basis using a non-linear function. The architecture is typically composed of an input layer followed by multiple processing layers. The more of these intermediate processing layers, the more complex relationships that the network can model. In this line, when the number of layers becomes large enough, the network is called a deep neural network [77], which also requires more careful training protocols. These intermediate layers are finally collected into an output layer which produces the aggregated result of all previous computations, and depending on the shape and type of processing performed throughout, the network is able to learn to output any value: from regression to classification, latent spaces, distributions, complex states, style transfers, *etc.*

While neural networks are potentially the most powerful machine learning model, their design, training, validation and deployment is plagued with difficulties. To mention a few:

1. The input data must be preprocessed, which includes typically procedures for normalization, outlier removal and cleaning, data augmentation, balancing, *etc.*
2. Neural networks require huge amounts of data in order to generalize properly.
3. Training times can be very long, especially as the size of the architecture grows since the number of weights does not scale linearly. For instance, the language model GPT-3 has over 175 billion parameters and training costs are estimated at several million dollars [73].
4. Each problem usually requires a different architecture, different set of data or examples (if supervised), training methodology, *etc.*
5. The resulting predictions of neural networks are hard to explain, despite the recent advancements in contribution analysis [78].

Despite all difficulties, steady research is being carried out in the field of artificial neural networks, with major stakeholders starting to invest, research and deploy ever more commercial solutions [79].

Chapter 3

Ultra-fast sequence comparison on single-processor systems

3.1 Methods

3.1.1 Introduction

The first contribution is a High Performance Computing application for the ultra-fast comparison of large-scale DNA sequences, namely “CHROMEISTER”. The article corresponding to the following work is available at [48]. The reasons that motivate this work are several, both from a computer architecture and computational biology perspective. On one hand, not every researcher (or students as well) has access to large, distributed computing infrastructure or expensive acceleration equipment, and even if it was available, more computation does not always imply more throughput in terms of results. This is the case more often than not in sequence comparison: sometimes the output of a sequence comparison experiment does not yield much valuable information, but nevertheless requires huge amounts of computation. Therefore, one of the main reasons for this proposal was the idea to design a fast algorithm that could be used as a screening-filter to discard unnecessarily expensive sequence comparisons, computing-wise.

On the other hand, from a computational biology perspective, the fact that sequencing whole genomes has become much more affordable (in fact at an exponential rate [80]), has resulted in databases being extended with more but also larger genomes. Thus, a need for high-throughput analysis has emerged, which combined to the previous idea of filtering comparisons, consolidate the necessity for a lightweight, high-performance application that can deal with massive sequences in reasonable time in order to save processing hours from posterior, exhaustive comparisons.

The code for this proposal can be found in the GitHub repository CHROMEISTER.

3.1.2 Formulation

In this section, a closer inspection of the distribution $H_{a,b}$ is presented, which corresponds to the set of matching words between two sequences previously discussed in subsection 2.3.1 "Definitions". In particular, the distribution is further extended into three other distributions, which enables to model which information is kept and which is discarded in order to allow for a lightweight processing algorithm. In particular, $H_{a,b}$ is split into:

- $W_{a,b}$ will be the distribution of random matches between sequences s_a and s_b . For sufficiently large $n_a = |s_a|$ and $n_b = |s_b|$ (the length of the two sequences, respectively), the probability of finding random matches is modelled as a binomial distribution $B(|C_a \times C_b|, \frac{1}{4})$, assuming uniform and equal frequencies. Therefore, $W_{a,b}$ is the distribution of random noise originated by simple chance.
- $R_{a,b}$ will be the distribution of any word-match with frequency higher than one, *i.e.* such that its words w_i, w_j are repeated more than once throughout sequences s_a and s_b .
- Lastly, $T_{a,b}$ will be the distribution that corresponds to word-matches of frequency equal to one between s_a and s_b , such that $T_{a,b} = H_{a,b} \setminus (W_{a,b} \cup R_{a,b})$. This is, seeds whose words (w_i, w_j) are unique and shared.

Notice that on one hand, the distribution $R_{a,b}$ will account for any type of DNA repeat such as tandem repeats, low-complexity regions, interspersed repeats, *etc.* On the other hand the distribution $W_{a,b}$ contains those words originated by chance and cannot be considered as repeats under biological examination. Of course, there will be some degree of overlapping between $W_{a,b}$ and $R_{a,b}$, however this does not affect the resulting distribution $T_{a,b}$, which, by definition, will contain only words that are neither part of the repeats nor the noise, thus containing only conserved signals.

A similar analogy for the distributions $W_{a,b}, R_{a,b}$ and $T_{a,b}$ would be the case of comparing two books written about a related subject but in different centuries. In such a case, and allowing some degree of variability, $R_{a,b}$ would contain the matches between highly repeated words (such as “the” or “you”), $W_{a,b}$ would contain the random matches product of incorrect typing and other errors and $T_{a,b}$ would contain the word matches that correspond to specific keywords related to the subject (such as “mars” or “Lagrange”).

3.1.3 Algorithm overview

In short, the algorithmic steps proceed by: (1) two DNA sequences are received as inputs; (2) a word dictionary is built based on the query sequence; (3) the words in the reference sequence are matched to those in the dictionary using inexact subsampling; (4) matched words are filtered based on uniqueness; (5) positional information is incorporated into a matrix representation and the local optimum is found; and (6) a scoring distance is assigned to the comparison, and Large-Scale Genome Rearrangements (LSGRs) are detected based on an iterative process. These steps are depicted in Figure 3.1.

The additional functionality depicted in Fig 3.1, namely the combination of generated outputs with other programs is shortly described as:

- Sequence comparison. Using GECKO it is possible to extend and refine the signal detected by CHROMEISTER in order to obtain more detail of a comparison that has already been generated, as opposed to starting it from scratch.
- Block tracing. An application that builds on top of CHROMEISTER to utilize the signal to perform tracking of large blocks of genomic DNA across multiple species.
- Analysis and classification. A set of scripts that were developed to perform automatic clustering based on the similarity of species.
- Visual inspection. A method to generate a visual representation of the comparison along with automatic detection of blocks.

3.1.4 Indexing

Although all stages in the proposed algorithm are strictly linear in the size of the inputs, the most computationally demanding one is the indexing of words. For this purpose, a customized hash table is used as indexing data structure. First, since no duplicate elements are stored, it is possible to ignore chaining mechanisms. Second, the value of k to index the hash table is limited to a prefix of size $k = 12$, corresponding to the first 12 nucleotides. The use of this value of k is twofold:

- The words to match are scanned in an overlapping fashion and therefore there is a potentially unique substring of size 12 within them such that it does not collide with other entries in the table.
- A table of size 4^{12} is manageable in terms of memory and requires less than 300 Megabytes if considering 8 bytes for the key and 8 bytes for the location of the word.

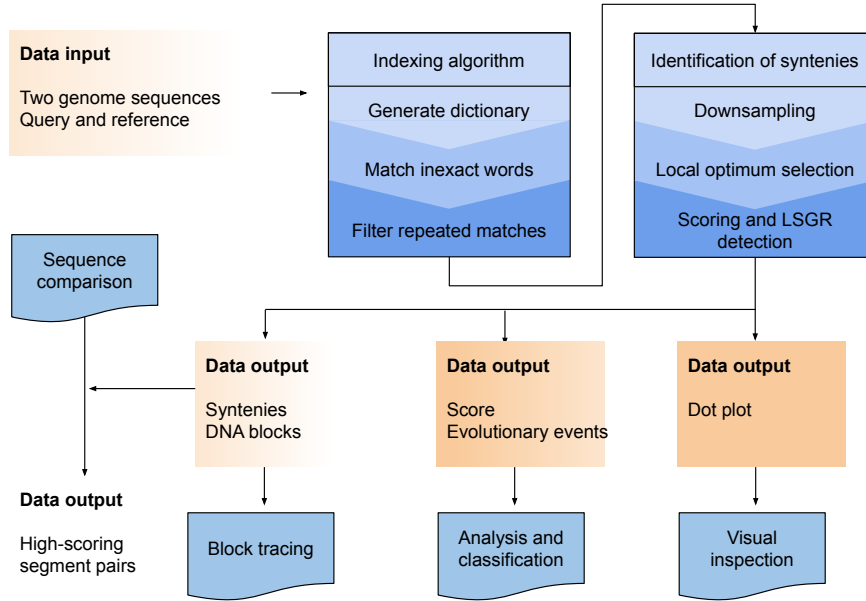


Fig. 3.1 Workflow of the proposed method and further functionality. The input data is fed to the algorithm and then the outputs can be either combined with other sequence-comparison-based approaches or as input to further experiments.

A second mechanism is devised to enable a certain degree of variability, since depending on the input data a fixed string of size 12 can be too restrictive to generate matches. For that purpose, recall the definition of the perfect indexing hash function (see Equation 2.2), which is now generalized and extended into the family of functions F , namely:

$$F = \bigcup_{z=1}^{\infty} g(x)^z \quad (3.1)$$

Where $g(w_{i,j}^k)^z = \sum_{j=1}^k (|S^{j-1}| * v(w_{i,j}^k) * belongs(j-1, z))$ and the function $belongs(j, z)$ is defined as:

$$belongs(j, z) = \begin{cases} 1 & \text{if } \exists x \in \mathbb{N} \mid j = z * x \\ 0 & \text{otherwise} \end{cases} \quad (3.2)$$

In short, $belongs$ returns 1 if j is a multiple of z , including zero in the definition of natural numbers. Thus, F represents the collection of functions that contain all possible indexing functions that are formed as a consequence of skipping the nucleotides whose positional

value is not a multiple of the given parameter z . For example, given $z = 2$, an alphabet of size $|S| = 4$ (as is the case in DNA strings), and a word $w = \text{TAACGT}$ the resulting function g applied would produce the following:

$$\begin{aligned}
 g(w)^{z=2} &= \sum_{j=1}^k (|S|^{j-1} * v(w_{i,j}^k) * \text{belongs}(j-1, 2) \\
 &= (4^0 * v(\text{T}) * \text{belongs}(0, 2) + 4^1 * v(\text{A}) * \text{belongs}(1, 2) + \dots \\
 &= 1 * 3 * 1 + 4 * 0 * 0 + 16 * 0 * 1 + 64 * 1 * 0 + 256 * 2 * 1 + 1024 * 3 * 0 \\
 &= 515
 \end{aligned} \tag{3.3}$$

In this particular case, only the nucleotides at positions that are multiples of two would be used to compute their hash. Note that 515 is the hash value that would be stored for word $w = \text{TAACGT}$ at the location corresponding to its prefix (smaller than $k = 12$ in this case). In essence, the collection of functions F contains all indexing functions $g(x)^z$ that to compute the hash of w use either all letters, half of the letters, a third of the letters, and so on (intermittently). That is, the higher the z value of the $g(x)^z$ function, the higher the number of collisions there will be in the hash table. Ideally, every skipped letter would increase the number of collisions by $|S| = 4$, since $w_{i,j}^k$ can take up to 4 different values at position j . Notice that for any given $z > 1$ the function $g(x)^z$ will produce an inexact (collided) indexing, which enables to find longer and imperfect matches (thus accounting for variability) while also incorporating positional information. Hence, any function from the family F can be used to enable a heuristic subsampling of inexact k -mer matches without requiring the actual computation that is usually needed. For instance, in a brute-force approach, retrieving a key for a word w while allowing one mismatch would require checking up to $3k$ locations in the hash table. Therefore, this procedure enables to retrieve inexact k -mers in $O(1)$ time using a hash table-based structure.

3.1.5 Matching

After the reference sequence has been indexed, each word w from the query is scanned against it. If two words w_i and w_j share the same prefix and partial hash, then they are considered a matching hit $h_{i,j}$. The cartesian position of $h_{i,j}$ is annotated into a matrix H_m which represents the sequence comparison. The size of this matrix is variable and user-defined, but restricted to be of much less resolution than the actual comparison. This is, if the comparison features two sequences of lengths l_1 and l_2 then the matrix should have sides of length $l'_1 \ll l_1$ and $l'_2 \ll l_2$. While doing so certainly reduces the “resolution” of the comparison (since

positional information is compressed), it also enables to group together regions of high concentration of hits.

To illustrate the effect of grouping regions, consider a signal that spawns along a sufficiently large enough section of a sequence comparison. The signal is composed of individual hits $h_0^{i,j}$ through $h_t^{i+l,j+l}$ which are scattered throughout a diagonal $d = i - j$. Without seed extension, these seeds might not be differentiated from random matches; however, when downsampled, their range from (i, j) to $(i+t, j+t)$ gets compressed within a single point (e.g. (i', j') now encapsulates the section (i, j) to $(i+t, j+t)$) and therefore the concentration of seeds rises at point (i', j') , for it now includes other seeds that followed in diagonal d . The same does not occur with random matches, for these do not have collinearity by definition.

In essence, this effect is similar to image downsampling: grouping hits closely together removes unwanted gaps between them (which are typically filled by gaps or misalignments), just as downsampling images reduces the number of unwanted features and noise while also making them more robust to rotation and translation, typically used in convolutions within Machine Learning applications.

3.1.6 Selection of local optima

After the compression step, the matrix H_m will contain the accumulated number of unextended seeds per cell. These values, which represent the raw concentration of seeds, need to be filtered furtherly for two reasons: (1) removal of random noise and (2) to ease up the posterior scoring and event detection mechanism. Regarding random noise, since repeats were removed as part of the indexing process, any data points that are left and do not correspond to a grouped diagonal are considered noise. These must be removed in order to not interfere with the selection of conserved signals which takes place afterwards.

The algorithm behind this process can be seen as finding the marginal distributions of local optima (*i.e.* the highest concentration of seeds per row or column). The marginal distribution is sought for since only one cell in a row or column can contain the highest concentration, and in fact any other concentration in that same row or column must be a repeat according to the framework described in subsection 3.1.2 "Formulation".

In Figure 3.2, three parts of the process of selecting local optima are depicted. The left part shows the concentration values for matrix H_m (omitting zeros) and highlights the values that will be discarded for being lower than their row or column maximum. Note that due to the size of this example, a certain degree of overlapping takes place in the boundary regions, but such is not the case in real-sized comparisons. The center part shows the selected marginal optima per column. Lastly, the right part shows a real comparison where dotted

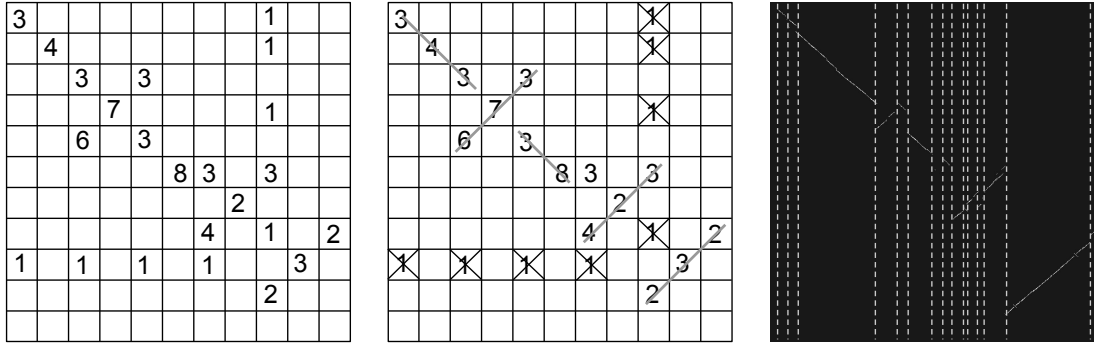


Fig. 3.2 Visual representation of the local optima selection. (Left) The matrix H_m (zero concentration cells have been omitted) highlighting repeats. (Center) The selection of marginal concentrations. (Right) Example of a comparison where marginal optima is selected by columns.

lines show the boundaries of the different blocks. Notice that no overlapping occurs (except for some small sections, which is still due to the size of the comparison).

3.1.7 Scoring of comparisons

Once all previous steps have completed, the resulting hit matrix H_m is a low scale representation of the two-dimensional density function of the inexact matching words between both sequences. In order to score the comparison, it must be determined what constitutes a positive contribution and a negative contribution. Signals comprise contiguous and adjacent nucleotides (or in string matching, similar letters organized in approximately the same order) and therefore they tend to group diagonally (in a cartesian space). Therefore, diagonally adjacent hits contained in H_m will be weighted positively whereas sparse hits will be weighted negatively proportional to their distance to other hits. To account for this scenario, the score of a comparison is modelled as follows:

$$d_{\text{raw}} = \sum_i^{l-1} \text{taxicab}((H_m(i)), (H_m(i+1))) \quad (3.4)$$

Where d_{raw} is the raw distance or score (unnormalized) of a comparison. Notice that for a comparison where the sequences are completely equal (*i.e.* a perfect diagonal would appear in a dotplot representation) the raw distance would be twice the length of the sequences (due to the taxicab distance not allowing diagonals). However, a condition is introduced not to add up distances to adjacent points in diagonals. In the other case, for completely different

sequences, there would be no points from which to travel. In that case, the maximum distance is added per each row, which equals to the length of the traversing sequence. Last, the inductive case: further away hits will incur in linearly higher distance and vice versa, which is the property sought after. Regarding normalization, since the maximum penalty that can be added per seed is the length of the traversing sequence, it is guaranteed that the worst score two sequences can attain is l^2 where l is the size of the lower representation matrix. Recall that this matrix is square despite the sequences being able to have other length ratios. Therefore the normalized score, d , is calculated as:

$$d = \frac{d_{\text{raw}}}{l^2} \quad (3.5)$$

Which in fact yields a score between $[0, 1]$ and produces a uniformly distributed range of outputs, as can be seen in the examples shown in Figure 3.3.

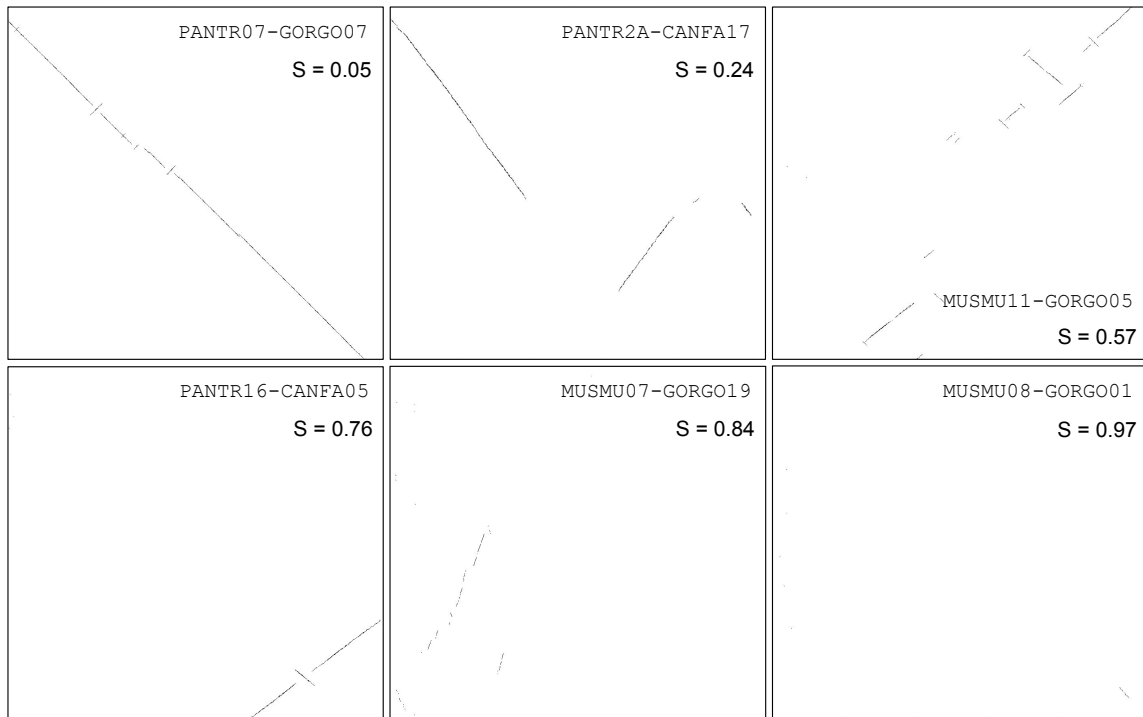


Fig. 3.3 Example of scored chromosome comparisons by the proposed metric. Comparisons are ordered by similarity from top-left to bottom right. Note that the angle of the alignments depends on the ratio between lengths of sequences, and therefore some variation from 45° can be seen.

3.1.8 Detection of evolutionary events

The detection of events is the last step in the processing workflow. The idea behind this step is to automatically classify blocks of matches as certain types of events, depending on their position and orientation. In fact, from a geometrical perspective, the problem amounts to detecting and classifying lines (for blocks are diagonal lines, *i.e.* adjacent word matches that group in a continuous way).

Recall that the aim of the proposed methodology is to generate an ultra-fast snapshot of a sequence comparison using as minimum features and resources as possible. Therefore, as a result of this process, the matrix H_m is not reversible, *i.e.* information is lost in the process, especially when it is compressed. This fact implies that the positional information of word matches h_{ij} is now an approximation and is bounded to the ratio of the lengths of the sequences to the size of matrix H_m . Although another approach is proposed to extend the value of these comparisons by retrieving and extending word matches in regions that are located nearby to conserved blocks (see next subsection 3.1.9 "Pipelined and streaming algorithm"), for the case of events detection, the algorithm must solely work with grouped blocks in the low-resolution matrix, and thus, it is treated as a classic computer vision problem where (1) lines need to be detected, (2) their length and orientation must be extrapolated and (3) classified according to a set of rules.

The Hough transform [81] is used for the detection of lines. Briefly, the algorithm proceeds by calculating the aggregated sum values of perpendicular distance and angle of each point in an image with respect to the origin. Therefore, a particular value of distance and rotation will accumulate a high score due to all the points that comprise it. Note that typically edge detection (such as with the canny edge detector [82]) is required as preprocessing in order to convert the input image to binary, such that each single point can only be either an edge point or not. This preprocessing is performed by the previous stage of selecting local optima, since it removes single points that are not part of larger blocks, which in turn represent line edges.

Lastly, reported lines by the Hough transform are evaluated and classified in terms of length, orientation and proximity to the main diagonal. Lines of length shorter than a given threshold will be removed. The ones that make it through the threshold will be classified either as:

- Conserved blocks: If their orientation is forward (45 degrees in respect to the origin) and their proximity to the main diagonal ($0 = x - y$) is less than the maximum distance (determined empirically as a fraction of the size of the matrix H_m).
- Inversions: If their orientation is reverse (-45 degrees).

- Transpositions: If their orientation is forward but their proximity to the main diagonal is further than the specified fraction.
- Inverted transpositions: The previous case but inverted.

Further reading on the types of rearrangements and events can be found in [83].

3.1.9 Pipelined and streaming algorithm

The compressed matrix H_m is irreversible, *i.e.* information loss occurs. This fact effectively limits the application scope of the algorithm in the sense that it becomes only useful to generate a fast preview, while further analysis is strongly limited. To overcome this problem, an extended pipeline is devised to stream sections of interest (which correspond to the local optima in H_m) to a second, more fine-grained sequence comparison program. With such a goal in mind, CHROMEISTER becomes a search-space pruning algorithm that is utilized prior to a second program in charge of actually generating the alignments. In short:

1. CHROMEISTER generates matrix H_m which contains the accumulated signal per cell (i, j) at a resolution of r bases per unit.
2. The DNA subsequence corresponding to each cell (i, j) containing a local optima (*i.e.* highest marginal concentration per row or column) is streamed to the fine-grained approach.
3. Alignments are produced per each subsequence comparison.
4. The resulting alignments are put together by concatenation and offset aggregation.

In this particular case, GECKO was employed for this stage since (1) it is highly modular, (2) has a low memory and CPU time consumption, (3) some of its mechanisms are implemented and extended for GPU architectures in the third part of this thesis and (4) task-level parallelism is available to speed up the comparisons.

A representation of the subsequence batching procedure is available in Figure 3.4 (the matrix H_m is simplified for illustration purposes). Each highlighted cell represents a marginal local optima for which maximum concentration of seeds has been found; and their enclosing subsequence is extracted and streamed to GECKO in order to compute the final alignments. The output of the process is shown in part (c) of the figure, where a zoomed-in region can be seen corresponding to the boundaries of the starting cell.

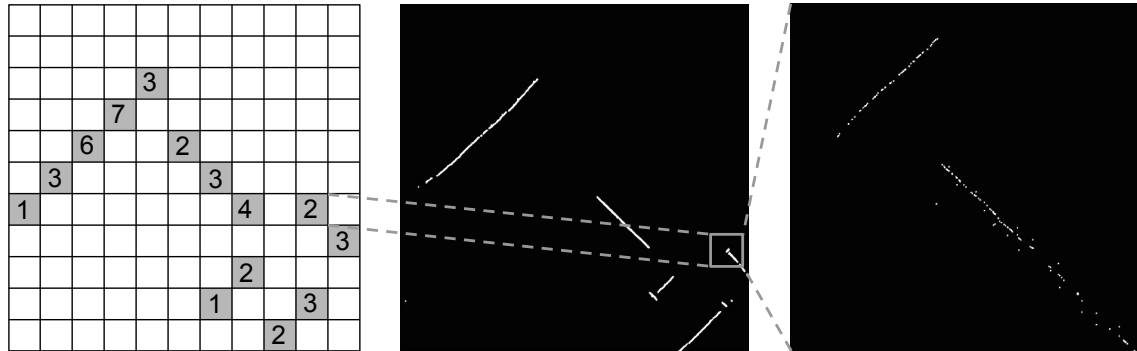


Fig. 3.4 Subsequence streaming from CHROMEISTER producing fine-grained results. (Left) The resulting matrix H_m along with the subsequence currently being processed. (Center) Section of the comparison corresponding to the subsequence. (Right) The actual section after performing fine-grained alignments using GECKO.

3.2 Results

3.2.1 Introduction

In the following sections, the proposed method for the ultra-fast comparison of whole genome sequences is tested against state-of-the-art methods in terms of runtime, memory consumption, usability and quality of results. All experiments depicted in this section used the following machine:

1. Local server: A server at the University of Malaga with 2x 8-core Intel Broadwell-EP E5-2620V4 processor at 2.1 Ghz, 64 GB of RAM and 2TB of local storage at 7,200 rpm. The version of GCC and g++ used to compile all binaries was 6.3.0.

Note that details of the file system are described since the runtimes in the following experiments also include the writing of results to disk for each of the programs in the state of the art comparisons.

3.2.2 Datasets

The following sequences were used for the experiments and comparisons described throughout this section:

1. A pair of plant genomes containing seven chromosomes each, namely the *Aegilops tauschii* and the *Triticum aestivum* genomes.

2. A massive dataset of all chromosomes belonging to the genomes *Homo sapiens*, *Pan troglodytes*, *Pan paniscus*, *Gorilla gorilla*, *Pongo abelii*, *Nomascus leucogenys*, *Papio anubis*, *Macaca mulatta*, *Macaca fascicularis*, *Chlorocebus sabaeus*, *Callithrix jacchus* and *Microcebus murinus*.
3. A set of additional chromosomes from five different mammalian species, namely *Homo sapiens*, *Pongo abelii*, *Mus musculus*, *Equus caballus* and *Bos taurus*.
4. A set of pairs of sequences containing *Bos taurus* Chr. 19, *Canis familiaris* Chr. 9, *Homo sapiens* Chr. X, *Mus musculus* Chr. X, *Brassica oleracea* complete genome, *Brassica rapa* complete genome, *Monodelphis domestica* Chr. 2, *Ovis aries* Chr. 1, *Aegilops tauschii* Chr. 1, *Triticum aestivum* Chr. 1, *Homo sapiens* complete genome, *Mus musculus* complete genome, *Aegilops tauschii* complete genome, *Triticum aestivum* complete genome.
5. A collection of bacteria sequences containing *Mycoplasma hyopneumoniae* 232, *Mycoplasma hyopneumoniae* 7422, *Escherichia coli* IAI39 and *Escherichia coli* O157.

3.2.3 Runtime and memory comparison

This section compares the performance of CHROMEISTER to other similar state-of-the-art software used for comparative genomics, and particularly for large genome comparison. These algorithms are CGALN [84] and NUCMER [85]. Times and memory consumption were measured ten times and averaged. Table 3.1 shows runtime and memory consumption for each of the methods using the dataset described in subsection 3.2.2 "Datasets" sorted by increased search space size.

As can be seen from Table 3.1, while CGALN and NUCMER are competitive in terms of smaller datasets, their time and memory requirements become unfeasible as the size of the input sequences grows. On one hand, CGALN is not able to finish the executions of the larger plant chromosomes and human and plant genomes in less than a hundred hours. On the other hand, NUCMER is able to complete such executions but it begins to require a much higher amount of memory (up to 50x more than CHROMEISTER) and an increased consumption of time. Finally, CHROMEISTER manages to use only constant memory throughout all executions and strictly $O(n)$ linear time.

Despite CHROMEISTER being useful to quickly determine similarity between massive sequences, it can also be used in conjunction with GECKO as a pruning mechanism (see subsection 3.1.9 "Pipelined and streaming algorithm") that removes noise and repeats, keeping the exhaustive comparison only to significant signals. Therefore, the comparisons performed

Table 3.1 Summary of the comparison executions. The second column indicates the size of the search space, which equals the product of the length of the input sequences. An asterisk in the NUCMER column indicates that it used more than one core in the computation, and thus the adjusted time is shown which equals the reported time consumption multiplied by the CPU usage. The time is shown in minutes whereas the memory is shown in Megabytes. The acronym “DNF” is short for “Did Not Finish”, and the flag is set when the execution has not finished after 100 hours of computation time.

Experiment		CGALN		NUCMER		CHROMEISTER	
Dataset	bp^2	Time (m)	Mem.	Time (m)	Mem.	Time (m)	Mem.
B19-C9	$4.03 * 10^{15}$	1.63	755	2.13*	812	0.89	994
HX-MX	$2.07 * 10^{16}$	5.93	2,032	5.14*	2,221	1.29	998
BC-BC	$1.18 * 10^{17}$	141.26	583	13.61*	5,814	2.42	1,022
M2-O1	$1.54 * 10^{17}$	17.04	3,618	15.14*	6,436	2.84	1,017
A1-T1	$2.57 * 10^{17}$	DNF	DNF	11.40*	8,357	2.84	1,022
HC-MC	$8.69 * 10^{18}$	DNF	DNF	168.86*	46,091	13.77	1,022
AC-TC	$1.39 * 10^{19}$	DNF	DNF	176.80*	52,021	15.96	1,022

with CHROMEISTER can be extended with GECKO to convert the coarse-grained signals into fine-grained alignments, which are of more interest for further biological research. The speed and requirements of the combined method is first analyzed followed by a comparison of the quality of the results to other methods.

Table 3.2 shows the execution times for the combined approach (from now on, pipelined execution) versus NUCMER. It is important to consider that in the previous analysis featuring the execution times for the different datasets (see Table 3.1), the parameters for NUCMER were tweaked in order to remove the extension of alignments (and thus make it fair compared to CHROMEISTER). In this case, the parameter was enabled to also produce the higher quality alignments and hence the running times are higher for NUCMER. It can be observed from Table 3.2 that the trend is similar to that of Table 3.1. NUCMER remains competitive in the smaller datasets, but quickly falls behind as the sequences grow. In the last dataset, NUCMER takes more than 100 hours and is therefore stopped. Still, the pipelined execution is able to maintain linear runtime and constant memory, mostly due to the fact that the search space is highly reduced by CHROMEISTER prior to the GECKO execution. Note that on dataset M2-O1 the execution time corresponding to the GECKO part of the pipeline is much smaller than expected. This is due to the running time of GECKO being proportional to the number of similarities found (number of word matches) and it is the case that M2-O1 has the lowest amount of the set, despite their medium sized sequence length.

Table 3.2 Performance comparison of NUCMER and the pipelined execution. In the case of the pipeline, the runtimes and memory consumption are reported for CHROMEISTER and GECKO respectively separated by a slash.

Dataset	NUCMER		Pipelined execution CHROMEISTER/GECKO pipeline	
	Time (m)	RAM (MB)	Time (m)	RAM (MB)
B19-C9	2.36	905	0.89 / 5.16	993 / 101
HX-MX	5.81	2,221	1.29 / 6.77	999 / 106
BC-BC	9.96	7,263	2.42 / 20.01	1,022 / 115
M2-O1	10.96	6,677	2.84 / 2.56	1,017 / 116
A1-T1	2,739.10	15,199	2.84 / 23.65	1,046 / 1,002
HC-MC	170.66	46,038	13.77 / 93.88	1,046 / 208
AC-TC	>6,000.00	53,500	15.96 / 212.35	1,046 / 1,314

Table 3.3 Coverage by method. The total alignment length is measured in nucleotides. The coverage percentage is calculated as the number of uniquely aligned nucleotides in the query divided by its length. GECKO is taken as the signal percentage reference for it attains the highest number of detected alignments. Values obtained for the dataset corresponding to the comparison between *Bos taurus* Chr. 19 and *Canis familiaris* Chr 9.

Method	Alignment length (bp)	Coverage	Signal percentage
GECKO	1,592,140	2.48%	100%
NUCMER	620,787	0.96%	38%
CGALN	369,071	0.57%	23%
MEGABLAST	705,900	1.10%	44%
Pipeline CHROM-GECKO	1,221,658	1.90%	76%

3.2.4 Coverage and quality comparison

The quality of the detected signal is now compared by a coverage analysis. In short, the alignments produced by the pipelined approach are compared to those of CGALN, NUCMER and also the standalone GECKO algorithm (without pre-filtering performed by CHROMEISTER). The program MEGABLAST [86] is also included due to its popularity in the Comparative Genomics scenario. For this experiment, the *Bos taurus* Chr. 19 was compared against the *Canis familiaris* Chr 9. Alignments were filtered with a minimum length of 300 base pairs and 70% identity. In essence, all algorithms agree on the alignments, with NUCMER however reporting more single scattered alignments which could be considered random matches due to the lengths of the sequences, however these are included in the coverage analysis for fairness.

Table 3.4 Confusion matrix for the traditional and inexact k -mer matching experiment. Rows represent truth values and columns represent predictions.

	Traditional exact k -mer matching		Inexact k -mer matching	
Actual/Predicted	Matched	Non-matched	Matched	Non-matched
Matched	7,712 (TP)	0 (FP)	8,680 (TP)	205 (FP)
Non-matched	2,001 (FN)	95,197,201 (TN)	1,041 (FN)	95,275,396 (TN)

Note from Table 3.3 that despite the pipelined approach losing nearly 20% of signal to the standalone GECKO (as expected, for many regions containing repeats and random matches are filtered out by CHROMEISTER on purpose), the approach still achieves higher coverage than any of the other methods.

Impact of the inexact matching methodology

In this section, the impact in terms of error rates of the inexact matching (see subsection 3.1.5 "Matching") is examined. Recall that inexact matching is an intended effect since it enables the method to account for slight mutations and small evolutionary events (whereas exact matching would not). For such a purpose, an experiment was set up where the first ten kilobases of the *Bos taurus* Chr. X were extracted, mutated at a rate of 1% per base and then the resulting k -mers were matched. Only unique words were used in order to understand the effect of the mutation in real sequences, *i.e.* in theory words could only match to their counterparts at the same position, however due to the mutation it is possible to observe how the inexact matching enables to match mutated words to their original counterparts (which would otherwise be not possible with traditional matching). This results in a simple classification process where four classes are observed:

1. True positive: A word is matched to its counterpart at the same position in the sequence.
2. False positive: A word is matched to any other word at any other position.
3. True negative: A word that is not matched to any other word except for its counterpart.
4. False negative: A word that is not matched to its counterpart.

The results of the matching of words are presented in Table 3.4. The experiment comprised 9,713 k -mers (10,000 minus the repeated ones) of size $k = 32$.

Table 3.4 can be explained as follows:

1. A higher number of true positives implies that more words were matched between the original and mutated sequences. In fact, the inexact method is able to match more

words since it allows mutated words to match to their original counterparts, whereas the traditional method would discard these.

2. No false positives can occur in the traditional matching (since the hash calculation is a bijective function, see subsection 3.1.4 "Indexing"). However, it is possible in the inexact matching since two different words can have the same hash. Nevertheless, this takes place at a rate of 2,05% words, which is low enough to be ignored.
3. The inexact matching achieves half the false negatives than the traditional matching. This is the desired effect since it implies that mutated words are being matched to their counterparts rather than being discarded.
4. Lastly, the number of true negatives is highly similar in both methods. Note that a true negative is counted for each word that is not matched to any other word except its counterpart (therefore there can be up to $9,712 * 9,712 = 94,322,944$). The differences between the exact and inexact methods are due to the 205 differences in false positives, since up to 9,712 true negatives can be detected per mutated word.

In conclusion, the inexact matching enables to improve the rate of true positives and true negatives at the expense of a small increase in false positives.

Impact of the uniqueness matching methodology

While the inexact matching is designed to allow for slight mutations, the uniqueness restriction is used to remove highly repeated elements that do not contribute to main signals. From a computational perspective, this translates into removing the most demanding process that conform the indexing of hashed words, *i.e.* the construction of the appropriate data structure (see section 2.3.3 "Data structures"). Nevertheless uniqueness comes at a price, namely a decrease in sensitivity, which will increase the number of false negatives since every match with over two repeats will be discarded. Therefore, to analyze the impact of this restriction, an experiment to count the number of discarded hits that belonged to a main, significant alignment (defined as an alignment which does not overlap with other alignments for at least 60% of its length) was performed. This filtering effectively removes repeating regions which are beyond the interest of the proposed method.

Three pairs of sequences were employed, namely two bacterial pairs (*Mycoplasma hyopneumoniae* 232 and *Mycoplasma hyopneumoniae* 7422, and *Escherichia coli* IAI39 and *Escherichia coli* O157), and one mammalian pair (*Homo sapiens* Chr. X and *Mus musculus* Chr. X). The inexact matching was disabled in order to not interfere with the uniqueness behaviour.

Table 3.5 The number of word matches and unique word matches detected in each of the datasets.

Method	<i>Mycoplasma hyop.</i>	<i>Escherichia coli</i>	<i>Homo sap. vs. Mus mu.</i>
All matches	417,633	1,802,818	15,691,528
Unique matches	14,440	57,704	15,152

Table 3.6 The number of unique matches contained within aligned HSPs. A minimum length was used to filter out small alignments.

Method	<i>M. hyop.</i>	<i>E. coli</i>	<i>Homo sap. vs. Mus mu.</i>
Minimum fragment length	100	200	300
Total aligned frags	324	556	1,037
Total frags containing unique hits	282 (87%)	543 (97%)	845 (81%)

Table 3.5 depicts the number of matches and unique matches per dataset. As can be seen, the number of unique matches is not strictly correlated with the number of matches. This is important for the theoretical limits of the maximum length of the sequences comparisons, *e.g.* if the sequences to be compared were longer than the total number of different 32-mers (4^{32}) then some truly unique matches would be removed even for probabilistic reasons. However since sequences are far from such lengths, it is safe to assume that unique matches are not inversely proportional to the length of the sequences.

The experiment is followed by counting the number of matches that belong to significant, non-repeated alignments (as described above). Table 3.6 shows the number of total alignments (HSPs) which contained unique matches.

In summary, over 80% of the found alignments contained unique hits. This number would be drastically reduced if repeating regions were included. However, the scope of the proposed method is to rather find the main signals and discard repeats to alleviate computational runtimes, as opposed to identifying repeating signals, which is a completely different subject (see [87]).

3.2.5 Massive comparison of complete genomes

The computation of the following synteny map of 66 comparisons concerning $\frac{1}{2}n(n+1)$ (with $n = 12$) full genomes took 23 hours and 21 minutes using only a single core and constant memory (similar to Table 3.1). On average, every comparison took less than 18 minutes despite the small consumption of computational resources. The synteny map is shown in Figure 3.5 "Similarity map between species". In essence, the map shows how

large sections of DNA still exist between species and how these have reallocated due to evolutionary changes.

It is beyond the scope of this manuscript to further discuss the biological results of the shown synteny map, nevertheless it serves as proof of what the proposed method is capable of producing with a reduced computational footprint.

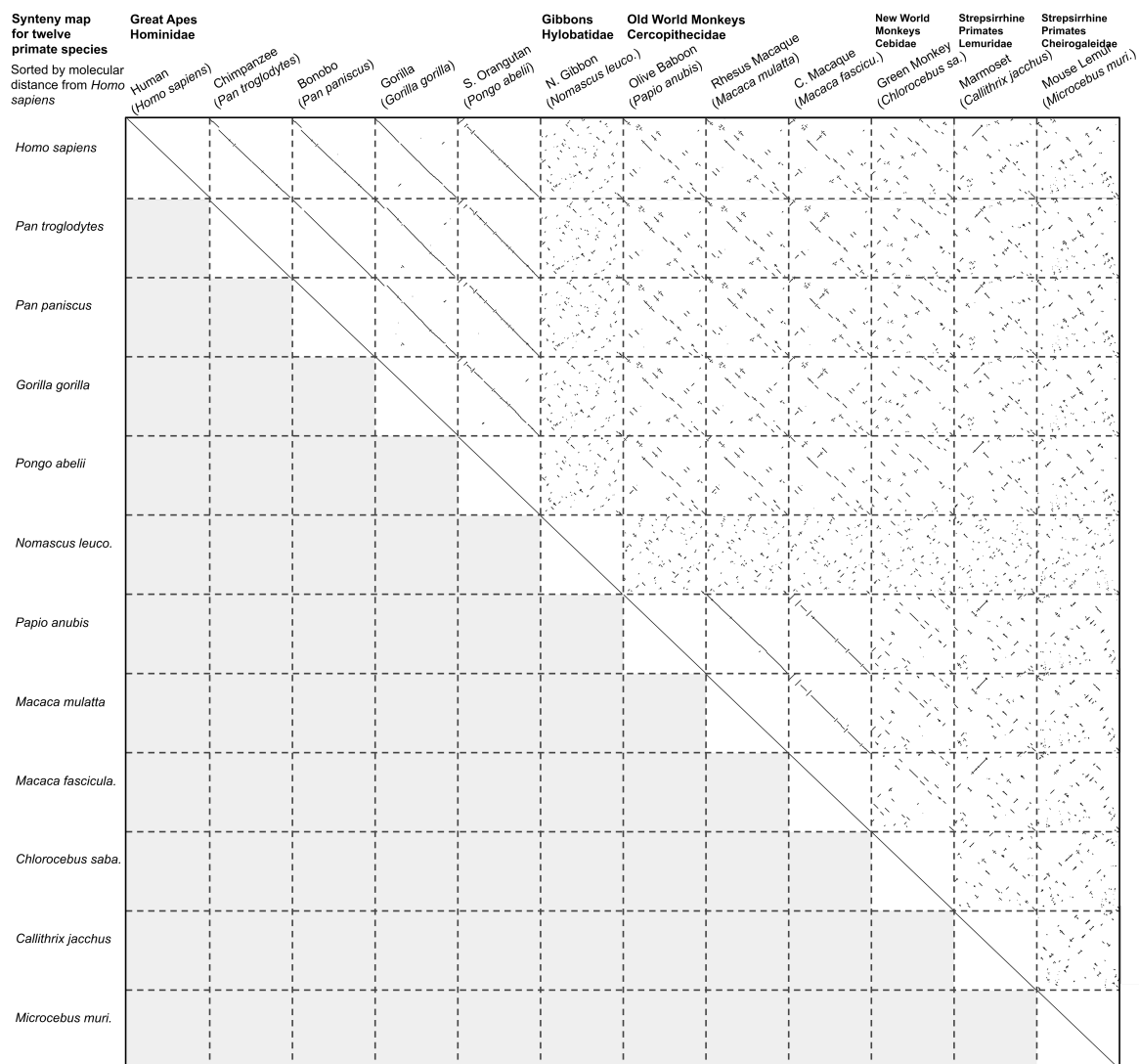


Fig. 3.5 Similarity map of twelve primate species as computed by the proposed method.

Chapter 4

Shared-memory multiprocessing for the fine-grained alignment of reads

4.1 Methods

4.1.1 Introduction

The second contribution is a parallel application for the fast and sensitive, fine-grained comparison of metagenomic data (or short sequences called “reads”). The proposed method is named IMSAME, which is the acronym for “Incremental Multi-Stage Alignment of MEtagenomes” [88]. The motivation to this work is twofold, namely (1) the difficulties of current methods to take advantage of the information contained within metagenomic samples and (2) the lack of parallel approaches to tackle the demand for computation in this type of sequence comparison algorithm.

Regarding (1), typical metagenomic comparison experiments include intermediary agents (databases containing curated and known sequences) in order to establish the mapping (*i.e.* map unknown sequences to known sequences based on alignments in order to infer the origin of the metagenomic sequences). However, using these intermediary and known sequences usually results in an arbitrary number of short sequences left without mapping, thus not taking complete advantage of the information contained in the metagenomic data. On the contrary, removing these intermediary sequences enables to find relationships between the data itself (without any bias), which can be revealed through techniques such as clustering.

Regarding (2), the problem lies in that metagenomic data has poor quality and therefore fine alignment methods are required, which in turn need much more computing power. The proposed method makes use of several, computationally-intensive stages for the calculation of alignments, each providing a different level of accuracy. To overcome these huge computa-

tional demands, the algorithm is parallelized using a coarse-grain strategy for shared-memory machines.

The code for this proposal can be found in the GitHub repository IMSAME.

4.1.2 Algorithm overview

The primary objective of this work is to be able to match reads (short sequences) together even if these present a high variability and/or number of errors. Nevertheless, to accomplish such a goal it becomes necessary to employ fine-grain alignment techniques, which of course incur expensive computation. Therefore, the processing pipeline is firstly divided into several stages, and secondly, parallelization techniques are applied to them in order to cope with the computational demand. The algorithm is divided in the following steps:

- A hash table is built in parallel for the reference metagenome and merged into a single dictionary.
- The query metagenome is loaded and split in blocks of different granularity according to a custom scheduler.
- A first set of blocks of work is given to all threads to start the computation of alignments.
- Each thread computes the matching of reads in the current block and the hash table.
 - Reads that include at least one match are extended with the optimization function described in section 2.3.3 "Seed optimization".
 - If the extension exceeds a minimum length, the read is then considered of interest and goes into the next stage.
 - Each read that has found at least one significant extended seed is fed into the bounded Needleman-Wunsch algorithm for the production of fine-grained alignments.
 - Lastly, aligned reads exceeding a percentage of similarity score are backtracked and reported.
- After finishing their current block, each thread asks the scheduler for the next block of work using a mutex mechanism until all blocks have been computed and no work is left.

The algorithm described above is depicted in Figure 4.1, where each step is shown along with the parallelism applied (shown as multiple threads).

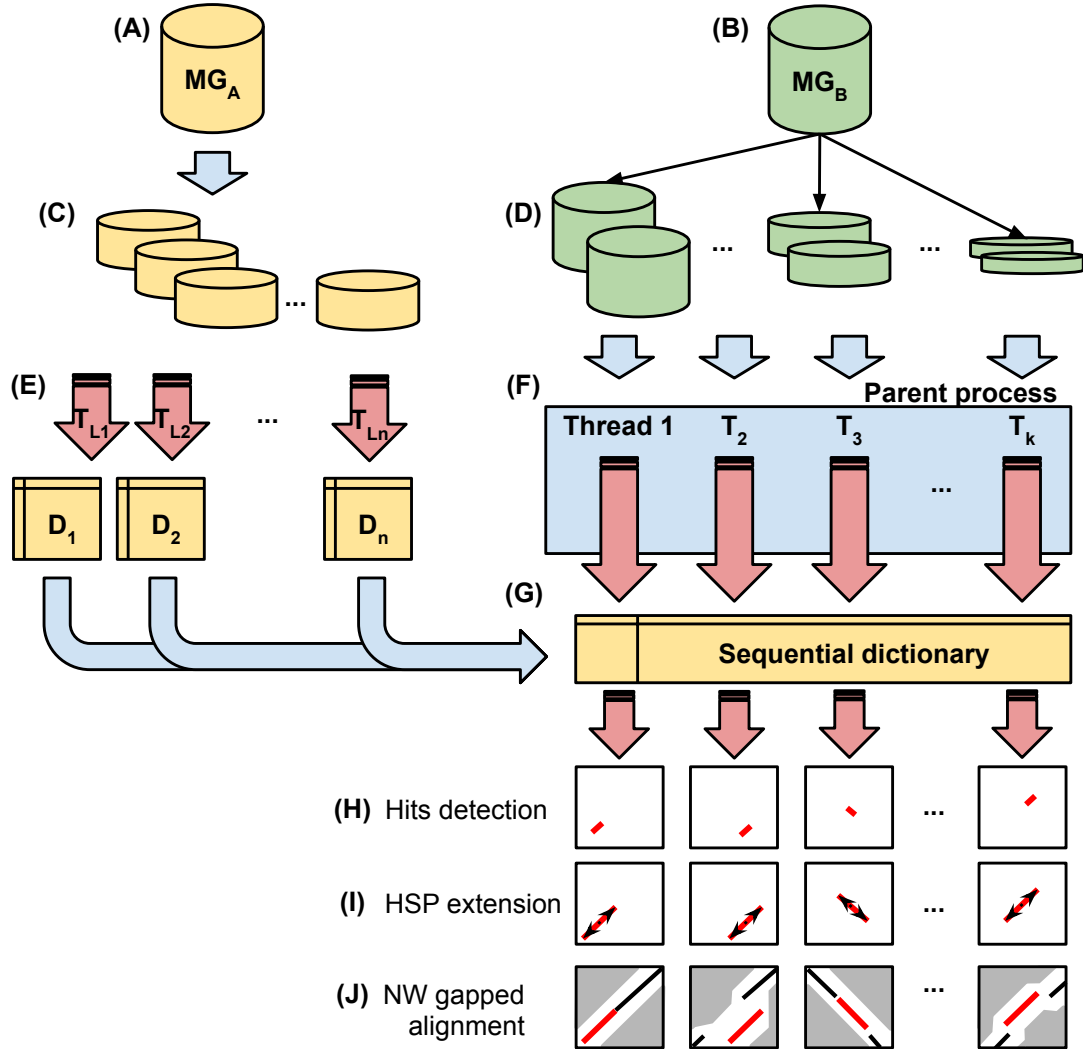


Fig. 4.1 Workflow diagram of the internals of IMSAME. (A and B) The workflow begins with two input metagenomes, namely query and reference. (C) The query metagenome is split in chunks to be loaded in parallel by n threads. (D) The reference metagenome is split using a guided scheduler. (E) Each loading thread builds a hash table dictionary. (F) Each split of the reference metagenome is fed to a different thread with variable granularity. (G) Each split is divided in words and matched to the corresponding dictionary. (H) At least one hit must be detected in order to advance to the extension stage. (I) The extension of the hit must yield a significant score in order to proceed to the gapped alignment. (J) Lastly, a bounded NW alignment is performed, including backtracking.

4.1.3 Indexing

The process of comparing both metagenomes starts with the indexing of the reference metagenome. The indexing step is required only for one of the metagenomes, since once such an index is available, the other input only needs to be queried against such index.

In a similar approach to the one employed in CHROMEISTER, a hash table of 12-mers is computed for the reference metagenome. The first difference with CHROMEISTER is that this table is a hash table for $k = 12$ and not for the prefix of size 12 of larger words, and the second difference is its handling of collisions through a chaining method. The reasons behind selecting $k = 12$ are several, from both the computational and biological perspective:

- Computationally:
 - To keep memory requirements manageable. In the parallelization stages, several hash tables are created, and the larger the k value, the more memory is required.
 - To remove user parameterization, particularly with the aim of avoiding fine-tuning by users and the removal of non-intrinsically related parameters (*e.g.* the percentage of similarity between two sequences is tied to the output results, whereas the k size is a heuristic parameter which relates to computational load). Using $k = 12$ fixes the problem of choosing parameters for non-experts and removes the possibility of unknowingly fine-tune the parameters to obtain the desired results.
- Biologically:
 - Providing sensitivity while retaining robustness [89].
 - The larger k becomes, the exponentially easier it becomes to miss candidates on which to perform a gapped alignment. This is especially true in the case of short and highly variable sequences (reads).

The creation of the hash table is performed in parallel throughout a custom number of loading threads. Each of these threads reads a different section of the reference metagenome and indexes words based on their 12-mer hash value. This parallel mechanism achieves two main benefits over sequential loading, particularly (1) threads can start processing while other threads are waiting for I/O operations (whereas with a sequential loading the only thread has to wait for the I/O operation to complete before processing) and (2) the hash tables are computed nearly in linear speedup in parallel if we do not factor in I/O operations.

The actual indexation process works by chaining repeated words in a linked list per each entry. Since the number of words per entry is unknown, a memory pool is created per each

thread and assigned as requested, therefore reducing the number of system calls for memory allocation.

Lastly, once all words are indexed in the hash tables, these can be merged (or concatenated) to create a single hash table. This is possible since the words in each entry were chained in order (according to their position in the sequences) and therefore simply linking the lists that belong to the same entry (although in different tables) is possible and maintains the consistency across the final table. The actual merging can also be performed in parallel, *i.e.* each thread takes care of linking the lists corresponding to a given section of the table.

4.1.4 Multiple-stage alignment process

After the merged dictionary is created, the query metagenome must be scanned and each word matched against the dictionary to initiate the multi-stage alignment process. These stages consist of (1) hash word matching, (2) linear extension and (3) gapped alignment. The complete process is performed in parallel and described in subsection 4.1.7 "Parallel implementation".

4.1.5 Word match and extension policy

The matching of words must be performed between every word in the query and every corresponding entry in the dictionary hash table. This can be an extremely computationally intensive process, even if parallelism strategies are applied. Fortunately enough, two filtering procedures can be used both in the linear extension and gapped alignment stages:

- In the linear extension, any word that belongs to a diagonal $d = x - y$ for which another word has already been extended in its vicinity, can be spared from extension.
- If an extension exceeds the minimum score and a gapped alignment is performed, any further hit can be ignored for the ultimate aligning step has already been completed.

While the matching of words is straightforward (calculation of the 12-mer hash and table lookup), the extension requires an optimization function and a condition to dictate when it should stop. In this case, the optimization function described in section 2.3.3 "Seed optimization" is employed. The ungapped alignment is run until the current aligning score reaches zero, which stops the ungapped alignment and reports the positions corresponding to the highest attaining scores.

4.1.6 Bounded Needleman-Wunsch

As described in section 2.3.3 "Dynamic programming methods", the Needleman-Wunsch algorithm is a dynamic programming algorithm which has $O(n^2)$ complexity both in time and space. Such complexity is very concerning especially when dealing with scaffolds or contigs, which are longer and can in some cases have similar lengths to fully assembled genomes. Also, while computing these NW comparisons in parallel alleviates time consumption, it increases the memory requirements proportional to the number of threads. Therefore, customizing the NW algorithm to satisfy the time and memory requirements becomes mandatory.

Keep in mind that any sequence comparison reaching the gapped alignment stage already guarantees a statistically significant ungapped alignment (produced in the previous stage). This means that the optimal alignment is likely to go through such extended alignment when aligning the complete sequences. Therefore, the proposal features a bounded NW algorithm which explores only a subset of the dynamic programming table as opposed to the whole thing. In Figure 4.2 (left) the bounded path is shown along with the candidate seeds and the original seed that began the alignment highlighted in red.

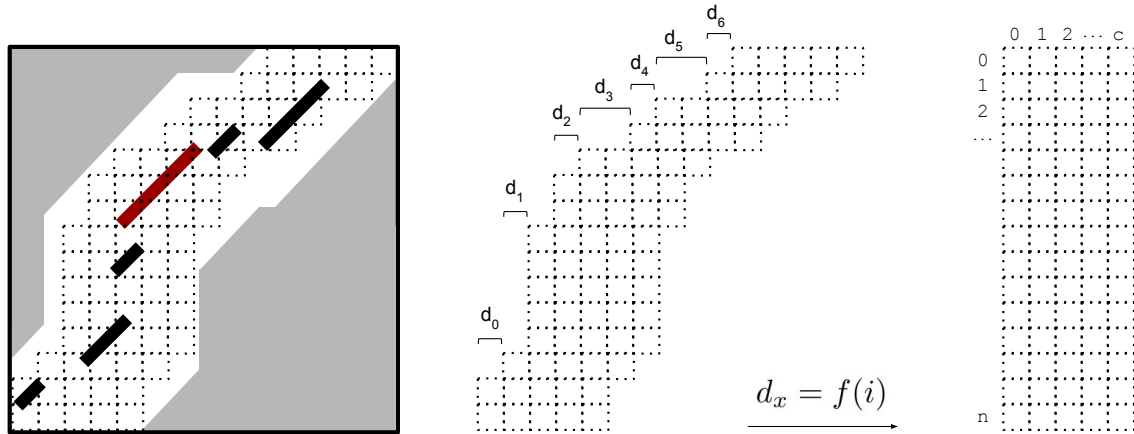


Fig. 4.2 The bounded Needleman-Wunsch procedure. (Left) Representation of the bounded regions explored around a significant ungapped alignment. (Center) The actual sections of the dynamic programming table explored, featuring the column shift d_x between adjacent rows. (Right) The subset of the matrix that is explored using the transformation function, which includes all rows but a fixed number of columns.

Notice from Figure 4.2 that only one dimension of the dynamic programming table is fully required (the vertical dimension in the plot corresponding to n), whereas the other dimension is bounded. Therefore the bounded NW uses only $O(n * k)$ time and memory with $k \ll n$, where n is the length of the query sequence and k is calculated according to the following equation:

$$k = \min(\text{window}_{\text{size}}, c * \sqrt{nm}) \quad (4.1)$$

The minimum operand ensures that the size of the window does not exceed the physical memory allocation. The second term, namely $c * \sqrt{nm}$ is the geometric mean of the lengths of query and reference sequences. The parameter c is simply a percentage between $[0,1]$ that acts as a reduction factor. The use of the geometric mean is to smooth out the size of the window in the case where the sequences compared have extremely different sizes, which can be a recurrent case in metagenomics.

The bounded path is calculated using Bresenham's algorithm [90], which enables to discretize a continuous line into its raster points. For each candidate alignment, the path is built from three points, namely the origin $O = (0,0)$, the coordinates of the extended seed $S = (x,y)$ and the ending point $E = (n,m)$, which comprise the two segments $\overline{OS}$ and $\overline{SE}$. Note that using the origin and ending coordinates forces the alignment to be nearly global, however this condition is set on purpose to better characterize similar reads (or the short sequences being used). Once the two segments $\overline{OS}$ and $\overline{SE}$ are discretized, each position of the kept dimension i of n is mapped to either an increment or decrement d_i depending on the direction of the segments.

Therefore when the actual alignment is computed, each coordinate access to the m dimension is replaced with the corresponding offset as dictated by d_i .

4.1.7 Parallel implementation

The proposal features parallelism both at the reference loading stage and in the computation of alignments. This implies that only the loading of the query metagenome is left without parallelization, for it merely requires to load all short sequences into an array and split them according to the scheduler. As opposed to the reference metagenome, it does not require to compute hash values, since these will be calculated in a per-thread fashion in the posterior stages.

Parallelization and granularity

IMSAME is designed to run one pairwise metagenomic comparison per shared-memory machine, and each of these machines employs a coarse-grained parallelism strategy. Moreover, if further metagenomes need to be compared, these can be simply distributed among other processors independently. Before jumping to the reasons why this parallelism approach was chosen, first recall how the algorithm is divided:

- A first step where the index dictionary is built.
- A second step where the matching, ungapped and gapped alignment are performed.

And now note that simply splitting both the query and reference metagenome to distribute them to different nodes is not possible since each short read in the query must be compared to the whole reference. Thus, a distributed approach requires either to duplicate work for one of the metagenomes or to copy (or rebuild) the index dictionary. While this is feasible, the potential for parallelism decreases significantly, especially if we consider that metagenomes can be variable in length and number and therefore it might result in not all the resources of a node being utilized.

In essence, the parallelization designed for IMSAME is based on splitting one task (*i.e.* the comparison of two metagenomes) among several threads using a coarse-grain job size where each one of the threads computes subtasks of variable size, as dictated by a scheduler to avoid load unbalancing issues. A fine-grained subdivision, such as computing one dynamic programming table by multiple threads would incur in loss of efficiency since (1) further synchronization mechanisms would be required (*e.g.* using an elastic pool readily available on request) and (2) the individual size of the gapped alignment might not be enough to compensate the overhead, since the parallelization of dynamic programming algorithms increases proportional to the size of the problem. In fact, fine-grained acceleration in dynamic programming sequence alignment is performed mostly on complete chromosomes or genomes, which are around 6 orders of magnitude longer than reads (*e.g.* 100 bp vs 100,000,000 bp).

Furthermore, in the case of shared-memory machines, there are other benefits related to memory usage:

- Only one copy of the query and reference metagenomes are kept.
- Only one copy of the sequential dictionary is required for the matching step.
 - The fact that the dictionary is in shared memory could lead to insufficient bandwidth and higher latency if all queries were performed at the same time by all threads; however this is not the case for most of the computing time is spent in performing both extensions and especially alignments, thus threads rarely lookup the dictionary at the same times.
 - Moreover, shared memory machines will often contain large L3 caches. As the number of cores utilized increases, so does the total L3 cache available, and this enables to partially (or even completely) cache the sequential dictionary since

it is shared among all threads. All threads will benefit from the caching of the sequential dictionary (which is also read-only, hence cache coherence is kept consistent), and can in fact incur in superlinear speedup (see results section). On the contrary, the sequences being compared can be cached in the smaller caches, for their use is thread and thus core dependent.

The memory benefits mentioned are related to the use of shared memory, especially the superlinear speedup. In the case of distributed memory, not only would these benefits disappear, but also additional network synchronization would be necessary.

Guided scheduler

A uniform distribution of jobs risks increasing the makespan of the algorithm due to the unpredictable size of each job or subtask fed to the parallel threads. This is, threads could end up idling while a single or few other threads are computing alignments, thus increasing the overall execution time or makespan. To avoid this scenario, a customized Guided Self Scheduler [91] is employed to vary the size of jobs as the processing time advances. The use of this type of scheduler is key to achieving a balanced load with low synchronization overhead. In this line, the query metagenome will be divided in batches of size b_i where i is the current processing step, and each batch comprises a pair of coordinates indicating the start and ending sequence in the query. The size of each batch is decreased by employing a variation of the $\frac{1}{x}$ family of functions, namely:

$$b_i = \frac{R}{M * t * i} \quad (4.2)$$

Where M is a parameter concerning the number of blocks and t is the number of threads in the parallel application. A block will comprise t batches of jobs, as can be seen in Figure 4.3.

The number of partitions M serves as a parameter to configure the grain size depending on the input size of the jobs. For instance, if comparing two read-based metagenomic sequences (*i.e.* small, short sequences) a small M can be used (less but bigger blocks), whereas for contigs or scaffolds (long sequences) a larger M should be used to create more but smaller blocks of work, thus reducing the risk of increased makespan due to unbalanced load.

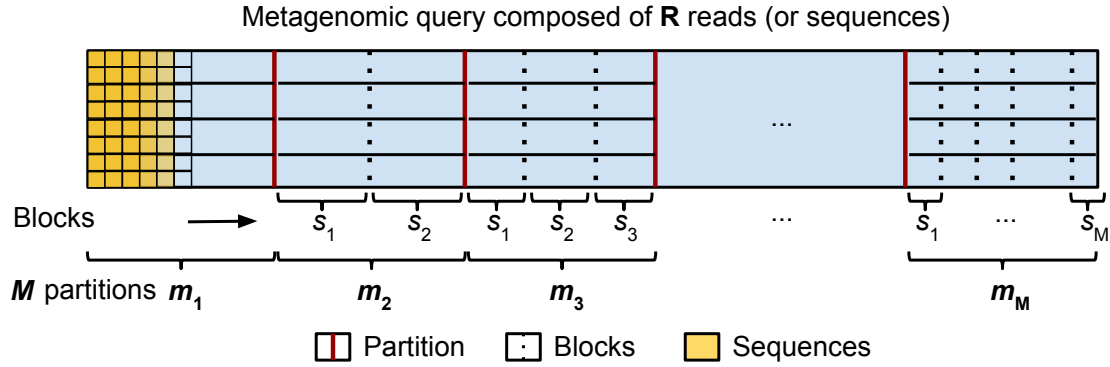


Fig. 4.3 Partitioning scheme for the workload in IMSAME. The input dataset is separated first into partitions of equal size, and each partition is separated into blocks according to the index of the current parent partition.

4.2 Results

4.2.1 Introduction

In the following sections, the proposed method for the parallel shared-memory comparison of short metagenomic sequences is tested against state-of-the-art methods in terms of runtime, memory consumption, usability and quality of results. In particular:

1. A sequential optimization comparison. This section evaluates the speedup that is the result of using the bounded-window dynamic programming approach (as opposed to fully exploring individual comparisons, see subsection 4.1.6 "Bounded Needleman-Wunsch") and the impact of the diagonal filtering (see subsection 4.1.5 "Word match and extension policy").
2. A parallel speedup comparison. Up to 64 cores are used to test the efficiency of the proposed method.
3. A state-of-the-art comparison with other software to validate the results of the method along with their parallel scaling properties.
4. A final discussion on the assessment and bias of metagenomic comparisons.

The executions described in subsection 4.2.3 "Sequential optimization comparison" made use of the local server (see local server in subsection 3.2.1) whereas the parallel executions described in advance made use of the following supercomputer:

1. Picasso supercomputer: The shared-memory fat nodes of the Picasso Supercomputer at the University of Malaga with 8x 10 core Intel E7-4870 processors at 2.4 GHz and 2 TB of RAM.

4.2.2 Datasets

The following metagenomic datasets were used for the experiments described throughout the Results section:

1. Three data sets belonging to the public database HMASM, part of the Human Microbiome Project [92], namely the *Anterior Nares* (SRS017697 and SRS019119), the *Posterior Fornix* (SRS064376 and SRS065347), and the *Bucal Mucosa* (SRS018359 and SRS057022).
2. The datasets TS1, TS2, SRS011397 and SRS053437 from the core gut microbiome study [93].

4.2.3 Sequential optimization comparison

In this section, the speedup in regards to the algorithm without optimizations is assessed for the overall and individual runtimes. Three experiments are considered depending on the nature of the sequences being compared: reads compared to reads (small sequences), reads and contigs (small and long sequences) and contigs compared to contigs (both long sequences). The datasets were obtained from the HMASM database (see dataset 1 and 2). Table 4.1 summarizes the distribution of reads and contigs employed in the datasets. The variability in length in the contigs is introduced knowingly in order to assess the speedup of the bounded window optimization (see subsection 4.1.6 "Bounded Needleman-Wunsch").

Each comparison was repeated ten times and averaged. The results are shown in Table 4.2, where not only the overall runtime and speedup are shown, but also the individual speedup corresponding to either of the optimized stages. Note that only one core was used to evaluate the performance impact of each strategy.

On one hand, the largest speedup is achieved in the alignments stage of the contigs to reads and the contigs to contigs dataset (columns with superscript a1). This is due to contigs being longer sequences (see Table 4.1), thus allowing the bounded window procedure to prune the search space at a larger degree. A smaller speedup is observed in the reads comparison (up to 4.32x), which is reasonable given that reads are smaller and less space can be left out of the anchored exploration. Among the contigs dataset, the contigs with reads dataset is able to achieve nearly double the speedup of the fully contigs dataset. This

Table 4.1 Summary of length distributions for the datasets used for the sequential optimization evaluation. Lengths are reported in base pairs.

Dataset	Type	# seq.	Min. length	Max. length	Avg. length
TS1	Metagenomic reads	217,386	38	482	239
TS2	Metagenomic reads	443,526	37	387	179
SRS019120	Assembled MGWS	40,761	306	51,219	562
SRS011397	Assembled MGWS	5,482	306	21,003	820
SRS053437	Assembled MGWS	4,843	306	7,147	801

Table 4.2 Performance comparison with and without optimizations. The columns represent, from left to right, the experiment, the total runtime, the runtime for the computation of alignments and hits and the speedup in both the alignments. All runtimes reported in seconds. The letter T stands for time, the suffix OR stands for the original, non-optimized algorithm, OP stands for the optimized version, and the superscripts a1 and h stand for the alignment and hit (matches) stages.

Experiment	T_{OR}	T_{OP}	T_{OR}^{a1}	T_{OR}^h	T_{OP}^{a1}	T_{OP}^h	S_{a1}	S_h
TS1 / TS2	3,142	1,712	1,868	830	431	812	4.32	1.02
SRS019120 / TS1	2,520	622	2,014	299	126	295	15.98	1.01
SRS011397-053437	665	78	657	4	71	4	9.27	1.10

occurs because the query contigs can be pruned to a width proportional to the size of the reads, whereas on the fully-contigs dataset the pruning cannot be as large.

On the other hand, the speedup resulting from the diagonal filtering of hits (columns with the superscript h) ranges between 1% and 10%. Although not much, recall that the diagonal filtering mechanism is aimed at (1) reducing the number of times a full alignment is performed, hence resulting in an increased speedup in the alignments stage and (2) improving robustness of the algorithm against low-complexity regions (such regions would cause two sequences to be aligned repetitively spawning from every adjacent low-complexity region). Moreover, the speedup due to the diagonal filtering is greater when both sequences being compared are long, since longer sequences are more likely to have several word matches than small ones, thus allowing the diagonal filtering to cancel many unnecessary alignments.

4.2.4 Parallel evaluation

The previous section featured a comparison of the sequential optimizations. In this section, the parallel performance and efficiency of the proposed method is evaluated using from 1 to 64 cores of the Picasso supercomputer. Three datasets were used corresponding to reads extracted from human body sites, part of the HMASM database (see dataset 2). These groups

Table 4.3 Runtime, speedup and efficiency comparison of the method for the three datasets ranging between 1 to 64 cores. Reported times are in minutes.

Cores	Small			Medium			Large		
	T (m)	Sp.	E (%)	T (m)	Sp.	E (%)	T (m)	Sp.	E (%)
1	21.65	1.00	100	740.72	1.00	100	5,170.59	1.00	100
2	10.75	2.01	100	369.58	2.00	100	2,677.19	1.93	96
4	5.31	4.08	101	184.87	4.01	100	1,310.41	3.95	98
8	2.63	8.22	102	92.27	8.03	100	663.08	7.80	97
16	1.35	15.94	100	43.89	16.88	105	313.96	16.47	102
32	0.73	29.47	92	22.80	32.48	101	159.94	32.33	101
64	0.42	51.03	79	11.94	62.00	96	82.14	62.95	98

of reads are separated due to their size in increasing order, totalling (1) 613,983 reads for the small dataset, (2) 3,401,514 reads for the medium dataset and (3) 14,295,910 reads for the large dataset.

As in the previous section, every execution was repeated 10 times and averaged to avoid system interference. The number of cores used was incremented in powers of two, thus each dataset was executed with 1, 2, 4, 8, 16, 32 and 64 cores, respectively. To reduce the computational runtimes of the experiment, only the forward strand was calculated.

For the most part of the small dataset the speedup is nearly optimal, however as the number of cores increases (besides 16) the speedup begins to drop as a result of not enough data to be computed in parallel to make up for the increased overhead (see Figure 4.4). The medium dataset is already large enough to show optimal speedup until the last test of 64 cores, where efficiency drops to 96%. As the large dataset demonstrates, the more data, the higher the efficiency becomes even when using 64 cores. Interestingly, all experiments exhibit superlinear speedup around the 4 to 32 cores mark, which can be seen visually represented in Figure 4.5 throughout the data points which are above 1.0 (*i.e.* 100% efficiency). This is due to the increase in cache memory available as a result of the Smart CPU Caching technology [94] of the Xeon E7-4870 processors in the Picasso supercomputer.

4.2.5 State of the art comparison

In the following section, two comparisons are performed between the state of the art methods, namely BLASTN [50] and BOWTIE2 [56], with the proposed method IMSAME. On one hand, the quality and quantity of the alignments produced by all methods is studied. On the other hand, the speedup and efficiency in a parallel setup is also analyzed. The same hardware setup employed in subsection 4.2.4 "Parallel evaluation" was used for both experiments.

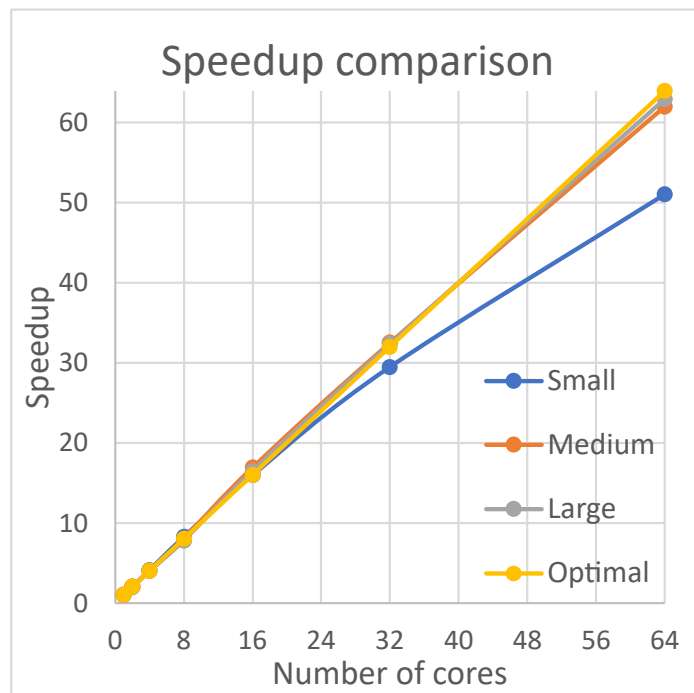


Fig. 4.4 Speedup plot of the shared-memory aligner for the three different datasets.

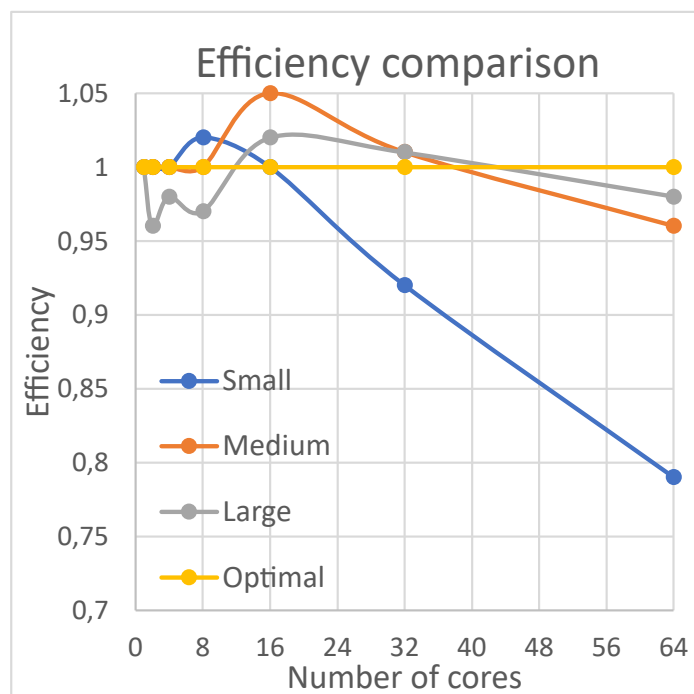


Fig. 4.5 Efficiency plot of the shared-memory aligner for the three different datasets.

Table 4.4 The number of alignments found by each method along with the average identity (%), coverage (length of alignment) and total runtime in seconds taken. Values calculated for the comparison between SRS023534 (Bucal mucosa) and SRS046973 (Anterior nares).

Method	Number of alignments	Avg. identity and coverage	Runtime (s)
IMSAME	11,972,580	86%, 97 bp	636
BLASTN default	9,353,752	97%, 83 bp	2,474
BLASTN tuned	11,232,871	96%, 83 bp	6,194
BOWTIE2 default	40,183	97%, 95 bp	14
BOWTIE2 sensitive	41,621	97%, 95 bp	23
BOWTIE2 tuned	5,280,060	96%, 89 bp	26,558

Quality of results

The quality of the produced alignments is tested against the well-known methods. It is important to remark that BLASTN is a general-purpose alignment tool not specifically designed to compare pairwise metagenomic datasets. In the same line, BOWTIE2 is a read-to-genome aligner, and thus not specifically oriented to pairwise metagenome comparison either. Nevertheless, no other state-of-the-art software was more suitable than the proposed ones. Each method was executed using four cores. Both the forward and reverse strands of the SRS023534 (Bucal mucosa) and SRS046973 (Anterior nares) of the HMASM database were compared (see dataset 1). The total number of alignments detected at a minimum percentage of identity and coverage (70% in both cases) for each method was measured. The results are shown in Table 4.4, which also includes the average alignment quality and the total execution runtime.

Several conclusions can be drawn from Table 4.4. In terms of the number of alignments, IMSAME is able to detect the most. When tuning the parameters of BLASTN to make it more sensitive, it is able to detect nearly as many as IMSAME, although at the expense of almost 10x the execution time. Also, the alignments detected by IMSAME are longer but of less identity compared to BLASTN. This is the result of the trade-off between optimizing either identity or length in the aligning function and thus none is more important than the other. In theory, by trimming off the additional length of the alignments reported by IMSAME it should be possible to increase the average identity up to the point of that reported by BLAST. Lastly, BOWTIE2 is able to find only a fraction of the alignments, although it does so extremely quickly when not tuned and at a high level of identity and coverage. However, when tuned to increase the level of detection (which still achieves only half of the number of alignments provided by both IMSAME and BLASTN), the runtime increases beyond reasonable, up to 4x that of the longest time taken by BLASTN.

In summary, IMSAME is able to detect the most alignments while requiring significantly less time than BLASTN and BOWTIE2. Regarding the average identity and coverage, all methods agree on the trade-off between identity and length as explained before.

Speedup and efficiency of the methods

The speedup and efficiency in a parallel execution scenario is studied for the three methods described beforehand. All methods were ran with exhaustive parameters in order to find the maximum number of alignments.

Table 4.5 shows the runtime (in minutes) and the speedup and efficiency of each method relative to the number of cores employed. Note that the runtimes corresponding to the building of the respective indices required for BLASTN and BOWTIE2 are not included, since these account only for few seconds to minutes.

The runtimes, speedup and efficiency of IMSAME are the same values as those depicted in Table 4.3. On one hand, while the sequential runtime of BLASTN is similar to that of IMSAME, its parallel performance quickly degrades when the number of cores goes above four. While this might seem surprising, it is in fact common knowledge that most of the BLASTN algorithm runs sequentially. On the other hand, BOWTIE2 has the highest runtime for single-core execution (as expected, since its primary use case is read to genome alignment, rather than read to read), nevertheless its efficiency remains between 60% to 70% for almost all steps. Especially interesting is the fact that the absolute efficiency in BOWTIE2 is initially lower for two cores than it is for four to eight cores. Moreover, in terms of relative speedup, BOWTIE2 has significantly higher speedup from the two cores in advance (*e.g.* relative speedup for two cores compared to one is approximately 1.33, whereas the following core increases have an average relative speedup of 1.94).

In summary, the speedup evaluation depicted in Table 4.3 along with the quality comparison described in Section 4.2.5 argue the case for the need of a fine-grained metagenomic aligner which has good parallel scaling properties, since neither BLASTN nor BOWTIE2 are specifically designed for read to read comparisons, and therefore their parallel implementations have a more difficult time in scaling properly with such type of data.

Furthermore, it is interesting to mention that other approaches to estimating the relatedness between metagenomic data also exist, such as COMMET [95] or SIMKA [96]. Due to their alignment-free nature, these methods are reliable for coarse-grained identification of species within metagenomic samples [97], but not for an accurate calculation of relatedness between samples themselves [98]. On the contrary, methods such as BLASTN or BOWTIE2 are able to offer the required level of fine-grained similarity, however at the expense of a slower processing rate, given their different goal. As shown, this causes difficulties, such as

Table 4.5 Parallel evaluation of the state-of-the-art methods. The abbreviations *IM*, *BL* and *BO* stand for *IMSAME*, *BLAST* and *BOWTIE2*, respectively. The first column for each method is the runtime in minutes, the second one is the speedup and the third one is the efficiency. Values calculated for the medium dataset used in Table 4.3.

Cores	IM. (m)	IM. Sp.	IM. Ef.	BL. (m)	BL. Sp.	BL. Ef.	BO. (m)	BO. Sp.	BO. Ef.
1	740.72	1.00	1.00	904.78	1.00	1.00	2611.61	1.00	1.00
2	369.58	2.00	1.00	585.51	1.55	0.77	1951.91	1.34	0.67
4	184.87	4.01	1.00	468.34	1.93	0.48	931.24	2.80	0.70
8	92.27	8.03	1.00	339.67	2.66	0.33	466.98	5.59	0.70
16	43.89	16.88	1.05	267.28	3.39	0.21	261.00	10.01	0.63
32	22.8	32.49	1.02	275.60	3.28	0.10	135.61	19.26	0.60
64	11.94	62.04	0.97	271.43	3.33	0.05	69.87	37.38	0.58

with the parallel execution when running with metagenomic data as both query and reference. Overall, this is the main reason on why the proposed method was developed, *i.e.* to enable fast, fine-grained and parallel comparisons of metagenomic samples without intermediate references.

4.2.6 Comparison assessment, bias and alignment results

The goal of *IMSAME* is to truly assess the similarity between two metagenomes without requiring hyperparameter tuning while also minimizing bias and errors that originate from their complex nature. It is certainly the case in many bioinformatics applications that changing of parameters hugely impacts the outcome of an experiment. To achieve independence from these practices, *IMSAME* features no computational parameters, *e.g.* the initial word match size is fixed at a small enough value (see subsection 4.1.3 "Indexing") such that decreasing it would not produce additional significant matches. The only parameters available are percentage of identity and coverage in regards to a gapped alignment, which do not affect the sensitivity of the program nor the execution time (as opposed to k size). In fact, these parameters are used to detect the sweet spot or turning point where bias and error are minimized by studying the analysis at different threshold levels which do not require rerunning the experiment, since global alignments provide all the information required.

On one hand, ROC curves with either the percentage of identity or coverage can be employed to analyze the effect of the varying threshold, and thus help determine the most appropriate value of identity or coverage that minimizes error and bias. On the other hand, clustering (such as k -means clustering [99]) at different threshold levels can provide different explanations of the underlying model, with none being the uniquely valid one.

The default output contains the alignments that exceed the specified threshold. This parameter might seem as a hyperparameter, however the best practice includes setting up to as low as possible (around 50%) so that there are as many alignment reports as possible. There are two considerations to this, namely (1) having an alignment report for each pair of reads enables to perform more detailed analyzes as explained in the previous paragraph and (2) the effect of lowering the threshold only incurs in a larger file size, *i.e.* more information is provided since it does not alter the original results: two reads aligning at 95% of identity will still align at that level. Therefore the percentages of identity and coverage do not alter the results, but rather serve as a threshold to control the amount of output information, which can be extremely large.

Besides the alignment file which contains all information, an aggregated metric, namely the Jaccard index [100], is provided to easily assess the similarity between two metagenomes. This index ranks the similarity of two populations between $[0, 1]$ based on the number of elements that are shared among both sets. In this line, the shared elements in IMSAME are those which exceed the identity and coverage threshold, which, once again, should be established through ROC curves or clustering analyses.

Chapter 5

GPU-acceleration for the irregular and exhaustive sequence comparison

5.1 Methods

5.1.1 Introduction

The third contribution is a GPU algorithm, GPUGECKO [101], for the heuristic and irregular sequence comparison. Sequence aligners that are implemented on GPU architectures are mostly restricted to the Needleman-Wunsch or the Smith-Waterman algorithm; not due to chance but rather because of the appropriateness of the two methods. As described in section 2.3.3 "Dynamic programming methods", they make use of a dynamic programming algorithm with *a priori* known size limitations, and therefore the amount of work is fixed and also known in advance, which enables programmers to take advantage of the massive parallel capability of GPUs due to the regularity of the algorithm. However, the complete opposite occurs in the case of this contribution, which features the heuristic seed-and-extend algorithm employed in GECKO but tailored to suit the workflow of GPU architectures. One of the particularities of this algorithm includes the fact that the number of seeds (or matching words) is not known until actual calculation, therefore creating load balancing problems which need to be addressed. It is potentially due to this and other difficulties (such as the heterogeneous granularity of the size of the alignments), that no other seed-and-extend algorithm has been implemented using GPUs.

In the next sections the integral parallelization approach will be described along with a description of the methodology employed and its link to computational biology.

5.1.2 The original GECKO algorithm

Since this contribution is the acceleration of a previously existing application, its internals will be briefly reviewed in order to understand the challenges that arise from transforming a CPU seed-and-extend algorithm into a high performing GPU algorithm.

The original GECKO algorithm is inherently sequential. It is divided into the following tasks (1) hashing of words, (2) sorting of words, (3) matching of words, (4) sorting of seeds, (5) filtering of seeds and (6) extension of seeds. The following dependencies between and within tasks can be identified:

- Hashing of words.
 - The hash calculation can be performed without any dependency.
 - The number of hashes is known (since it equals the number of words).
- Sorting of words.
 - There is a direct dependency with the previous stage. The sorting of words is useless unless all the hashed words of a given boundary are included.
 - Sublists of words can be sorted in parallel with the obvious requirement of a posterior merge [102].
 - The number of words remains the same after sorting.
- Matching of words into seeds.
 - There is a direct dependency with the previous stage and the algorithm cannot proceed until the words corresponding to a given boundary of the two sequences are sorted, or otherwise unmatched words would decrease the accuracy of the algorithm.
 - The number of seeds is unknown and ranges between $[0, \sum w_{i,a} * w_{i,b}]$, where $w_{i,x}$ are the number of words with hash i in sequence x . This has a clear quadratic upper bound in the size of the words.
- Sorting of seeds.
 - There is a direct dependency with the previous stage, since the sorting cannot begin until all seeds have been identified.
 - The same parallel merge requirements as in the sorting of words applies.
 - The number of seeds stays the same after sorting.

- Filtering seeds.
 - There is a direct dependency with the previous stage since filtering cannot be performed if the seeds are not completely sorted by diagonal value in a given boundary region.
 - There are internal dependencies since a seed s_i cannot be filtered out unless all seeds s_j who share the same diagonal $d = x - y$ and that are at a sufficiently close distance have been evaluated before. Therefore, to discard s_i it is needed to find the intersecting indices corresponding to close and diagonal-sharing seeds. Furthermore, unfiltered seeds need to be merged afterwards.
 - The remaining number of seeds after filtering is unknown and ranges between $[\min(1, s), s]$, *i.e.* there can be at most as many seeds as in the previous stage, and only one in the minimal example.
- Seed extension.
 - There is a weak dependency with the previous stage, for each seed unfiltered can be extended without requiring all seeds to have been filtered at the same time.
 - There are internal dependencies since the extension process of one seed can partially invade the extension process of another seeds, therefore crossing up and creating duplicate alignments. Synchronization is therefore needed in some form to avoid duplication.
 - While the number of extended seeds remains the same as the original number of seeds, their length is arbitrary and unknown, and ranges between $[0, \min(n, m)]$ where n, m are the length of the input sequences, respectively.

As can be observed, mostly all steps incur in some type of data dependency, either completely as a task or partially within subtasks. In essence, these dependencies translate in the following set of challenges, which need to be tackled from a GPU perspective:

- Large input sequences might require splitting and processing in batches. This translates into performance penalties from the impact of host-to-device data transfers.
- The number of seeds is unknown and quadratic in the number of words. Therefore the load can be unbalanced (*i.e.* some threads creating several orders of magnitude more seeds than others) and may result in memory overflow.

Table 5.1 Average proportion of the processing time and deviation along with the upper and lower confidence intervals at $\gamma = 95\%$ spent at each stage of the GECKO algorithm.

Processing step	μ time (%)	σ time (%)	Lower (%)	Upper (%)
Reverse complement	2.16	1.16	1.44	2.88
Words dictionary	27.83	14.37	18.92	36.73
Words sorting	27.08	13.66	18.61	35.55
Seed generation	10.22	7.02	5.87	14.57
Seed sorting	21.65	23.56	7.05	36.26
Seed filtering	1.63	1.32	0.81	2.45
Seed extension	9.44	6.61	5.34	13.54

- The length of final alignments is unknown and arbitrary. This implies a heterogeneous job size granularity whose balancing is unpredictable since threads can be left idling while others are still extending alignments.
- Seeds may overlap with each other when they are extended, amounting to duplicate computation and duplicate results. Therefore synchronization mechanisms are needed to avoid redundant computation.

Each of the above challenges are addressed in the following sections.

Runtime analysis of GECKO

A profiling analysis was carried out to analyze the computational requirements of each stage. Table 5.1 shows the profiling analysis of the original GECKO algorithm computed over the comparison of a dataset consisting of several species (see subsection 5.2.3 "Datasets").

Based on the profiling analysis, it can be concluded that the generation of the words dictionary along with both sorting procedures (words and seeds) take the most processing time in the original GECKO algorithm. This implies already a speedup benefit since sorting is arguably one of the most researched algorithms for GPUs [103]. Nevertheless, it is important to understand that all stages require acceleration, given that the upper bounds of these procedures can be quite high and consequently Amdahl's law [104] comes into play. In fact, it becomes mandatory to contextualize the values presented in the profile analysis. For example, when comparing chromosomes of *e.g.* mammalian species, the sorting step of the CPU-based GECKO might require around 100 minutes, and therefore the extension of seeds can require between 10 or 20 minutes. While this is already an impactful percentage of the computing time, the situation is aggravated if an all-vs-all comparison is factored in, where typically 400 comparisons are needed per each pair of genomes. In such cases, even 5% to

10% of the computation time results in large computing times that are unacceptable when considering a GPU-accelerated method.

5.1.3 Subsequence batching

As explained in section 2.3.2 "Number of patterns" and section 2.3.2 "Length of patterns", the number of seeds and the length of the alignments are unknown in seed-and-extend algorithms. From these two, it is the generation of seeds which must be handled more carefully, for it can cause memory overflows in current GPU devices due to its unpredictable and quadratic behaviour. A simple technique to cope with the explosion of seeds is to divide the complete problem in batches and compute each independently. However, on one hand, the smaller the batches, the less the performance (due to increased number of transfers, kernel launches, *etc.*), and on the other hand, the larger the batches, the higher the risk of overflowing the memory limitations due to seed explosion.

This seed explosion is due to the seeds being quadratic in number in terms of the number of words (*i.e.* absolute length of sequence or batch). This means that for a number of fixed words n_{words} there will be approximate a quadratic relation of the form $n_{seeds} = (n_{words} * s)^2$ where s is a similarity coefficient which models how related the input sequences are. Due to the unknown value of the s coefficient, it becomes difficult to predict the number of seeds that will be spawned in any comparison.

In the original GECKO implementation, secondary storage was employed to store seeds until processing time. When needed, these were fetched from disk to main memory, and although this approach enables nearly any computer (since sufficiently large disk storage is usually available in any common desktop workstation) to run virtually any sequence comparison, it also has a negative impact on performance. Such performance drawback is mostly due to two things, (1) writing and fetching from secondary memory is very expensive compared to main memory and (2) a collateral effect of massive reads and writes to hard disk drives is that it can result in inefficient parallelization, since these operations can block each other's processes, which might result in the starvation of even unrelated processes.

The depicted scenario shows the problems of using secondary memory as a means of storing seeds in the original GECKO implementation. However, in the proposed GPU implementation, using secondary memory is not an option due to performance drawbacks, and thus the subsequence batching mechanism is foreseen. Since it is not possible to calculate the seeds corresponding to a batch without going over their sorting and matching, a careful division in the size of subsequences must be chosen in order to not impact performance but also keep the number of seeds manageable. Figure 5.1 shows the subsequence batching

procedure employed. Note that the resulting alignments for each subsequence can be merged without losing sensitivity in the signals, since these are simply a linear extension.

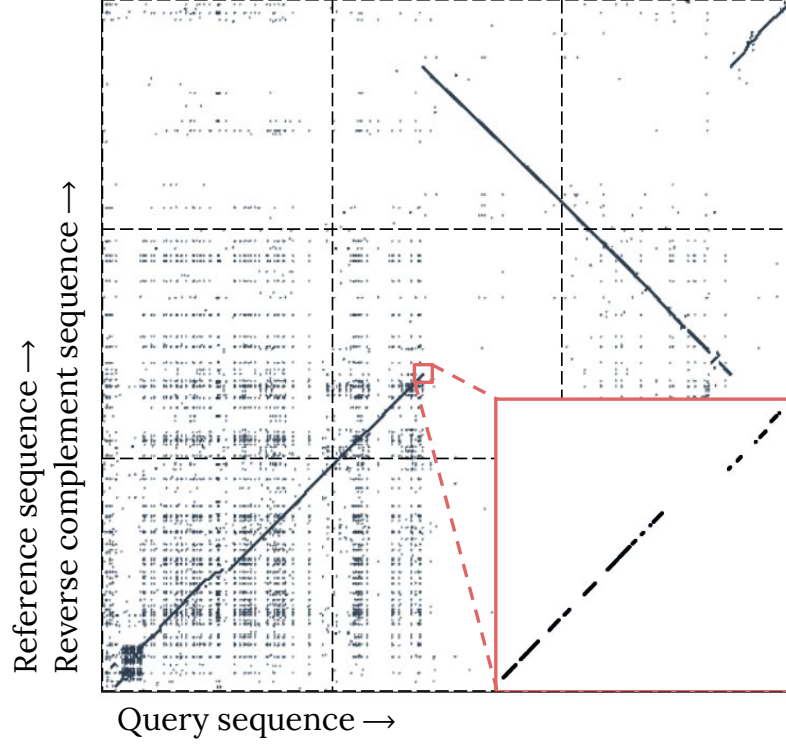


Fig. 5.1 Sequence comparison example using the subsequence batching procedure. The query and reference sequences are represented in the x and y axis respectively (the reverse complement of the reference is shown in the y axis as well). Each sequence is split into subsequences and computed independently (as denoted by dotted lines). The zoomed section shows how the main signal is decomposed into much smaller signals. The sequences employed for this example are *Gallus gallus* (common chicken) chr. 18 and *Meleagris gallopavo* (common turkey) chr. 20.

The size of the batches is managed by including a factor parameter which models the relationship of the s coefficient with the number of seeds. This parameter is fixed at a value (between 0.1 and 0.2 depending on the physical memory) that has been tuned experimentally in order to enable comparisons to run without errors while achieving adequate speedup. In this line, the size of the subsequence is therefore defined with the n_{words} parameter which depends on the available memory of the device as follows:

$$n_{words} = (\text{factor} * \text{global}_{RAM}) \div (8 * 2 + 4 * 2) \quad (5.1)$$

$$n_{hits} = ((1 - \text{factor}) * \text{global}_{RAM}) \div (8 * 2) \quad (5.2)$$

Where *factor* is the proportion of the memory of the device reserved for words (given *factor* in the range $[0, 1]$ and therefore $1 - \text{factor}$ is the amount reserved for seeds). The parentheses comprise the size of the data structure required for the sorting (8 bytes per key to store the diagonal value twice due to auxiliary storage needed). The use of this mechanism prevents the algorithm from overflowing due to the number of seeds. In fact, that is the case of other state-of-the-art algorithms (see subsection 5.2.5 "State of the art comparison").

5.1.4 Pinned memory

Using a batching procedure inevitably penalizes the performance due to additional data transfers between host and device. This occurs as a result of each subsequence of the comparison being run as a completely independent comparison, hence requiring several data transfers from and to the device. To mitigate the impact on performance of these transfers, pinned host-memory is used in the main memory of the host which allows the device to bypass the CPU and thus fetch memory pages directly through the Direct Memory Access (DMA) interface. In order to take the maximum advantage of pinned memory, a large block of memory is allocated at the start of the algorithm and managed internally to serve as a memory pool for each of the stages of the processing pipeline.

This enables to replace any system-memory allocations calls (which are slow due to memory fragmentation, context-switching, *etc.*) with conventional arithmetic operations between the pointers to the memory pool. The block of pinned memory allocated is calculated *a priori* in order to avoid reallocation calls. For a device of 4 GB of main memory, the pinned memory that needs to be reserved in the host is approximately 5 GB. This means that a naive scaling of the application with a GPU of 24 GB of RAM would require nearly 50 GB of pinned host memory, which would be completely unfeasible. Nevertheless, as will be shown in the following sections, the algorithm is further prepared to handle the execution requiring less than 20 GB of memory.

5.1.5 Recomputation in subsequence comparisons

While it might seem that the subsequence batching only splits the work in smaller pieces, it also incurs in slight recomputation or duplication of work. This occurs only in the stages corresponding to the hashing and sorting of words. Recall that the subsequence batching procedure divides the whole comparison into a grid of individual comparisons (i, j) which ranges from $(0, 0)$ to $(\frac{|S_1|}{n_{\text{words}}}, \frac{|S_2|}{n_{\text{words}}})$. This means that all the comparisons belonging to the same row (or column, depending on the stride dimension) will inevitably share the same words in the opposite sequence throughout the comparisons. For example, the first row $i = 0$

features comparisons $(0, 1), (0, 2), \dots, (0, \frac{|S_1|}{n_{\text{words}}})$, thus all comparisons $(0, j)$ share the same words in the S_2 subsequence. Also notice that this does not occur for the generation of seeds or their extension, since these depend on both the words from the S_1 and S_2 sequence.

This implies that duplicate work for the hashing and sorting of words is computed, being proportional to the number of subsequence divisions. For comparison purposes, a GPU device with 4 GB of global memory will subdivide sequences in batches of size $n_{\text{words}} = 25,000,000$ (see calculation formula in the previous section) which means that for regularly sized chromosomes (around 100 Mbp) a recomputation factor of 4 times can be expected per word in one of the sequences (which only applies to the hashing and sorting). While this recomputation can be avoided easily by storing the sorted words of each individual comparison (i, j) in the memory of the host and then simply reusing them when required, this approach incurs in additional memory requirements in the host which can, on average, increase pinned memory by an additional factor of 1, thus requiring approximately twice the pinned memory as in the approach without reutilization of sorted words. Nevertheless, the acceleration is not as significant as the increase in memory requirements due to the high computing capabilities of GPUs.

5.1.6 Algorithm stages and kernel implementations

In this section, each step of the algorithm is described. The computational reasons behind the implementation of each kernel are presented and discussed in terms of performance.

A depiction of the different processing stages and their relationships are represented in Figure 5.2. Notice that all blocks, with the exception of the striped one, are completely processed within the GPU accelerator. The description of the kernels will follow the processing order as shown in Figure 5.2.

Words kernel

The first step in the processing pipeline is the computation of words. The complete sequence is divided in overlapping substrings of size 32 and a numerical value is calculated for each one, which serves as a hash identifier. The hash value of each word is accompanied by its position in the sequence; thus building a key-value pair array. The complete data structure corresponds to the dictionary of the sequence, which still needs to be sorted in order to be efficiently queried.

Blocks of threads (of a multiple of the warp size) are launched to calculate consecutive 32-byte words (corresponding to 32 nucleotide words). For each word, a thread of the block must add the 32 arithmetic aggregations which depend on the value of the nucleotide itself

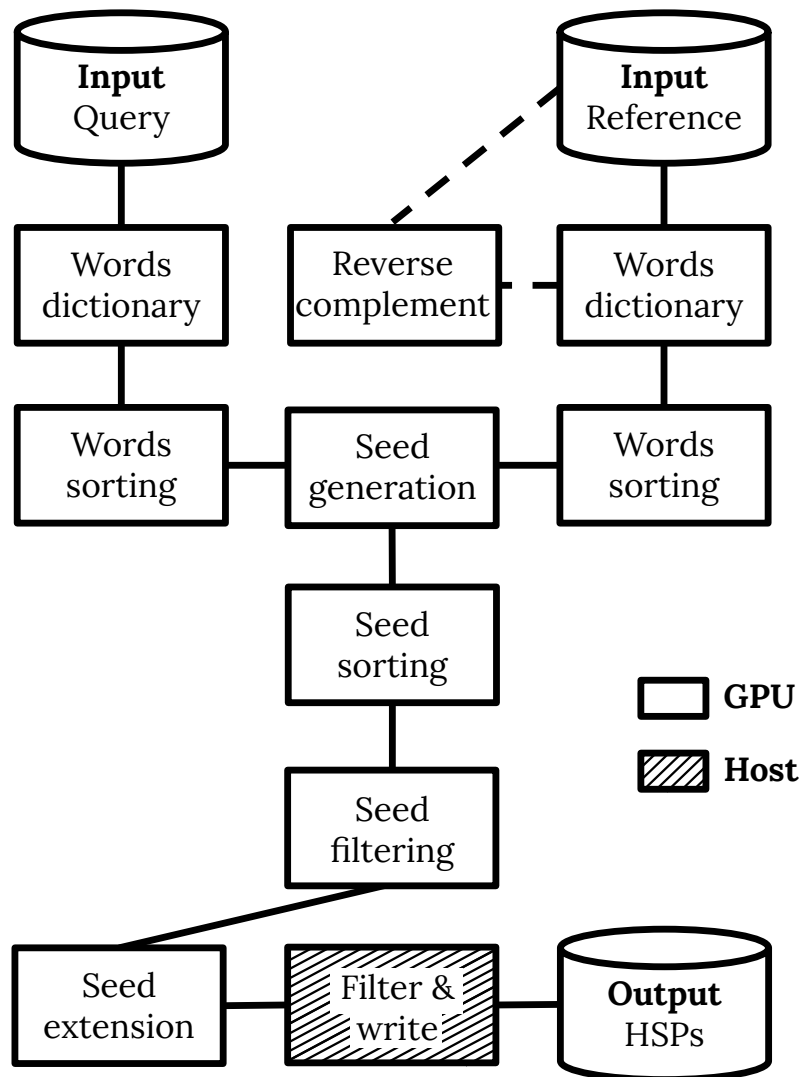


Fig. 5.2 Workflow diagram of the GPUGECKO algorithm depicting which kernels have been implemented within the accelerator or in the host.

(see section 2.3.2 "Difference in Alphabets"). These arithmetic operations consist in essence in a sum and a multiplication; however several implementation possibilities exist, depending on the type of operations used, namely (1) purely arithmetic, (2) predication with bitwise operations or (3) predication with cached constants, among others. The arithmetic-based algorithm uses several operations in order to remove conditionals and avoid branch divergence, as exemplified in Algorithm 1. Nevertheless, in most cases predicated conditionals can be used to avoid divergent behavior.

Algorithm 1 *k*-mer computing algorithm without branch divergence

```

1: procedure COMPUTE_KMER(String kmer)
2:    $h \leftarrow 0$ 
3:   for char  $c_i$  in kmer do
4:      $v \leftarrow (c_i \& 6) \gg 1$ 
5:      $h \leftarrow h + (1 \ll (2 * i)) * v$ 
6:   end for
   return  $h$ 
7: end procedure

```

Moreover, as can be observed from Algorithm 1, up to 6 arithmetic operations are required to carry out the hash calculation, which unfortunately incurs in high instruction dependency and thus lowers the instruction per cycle count. In the case of the predicated bitwise operations, the algorithm proceeds by incrementally bit shifting the current hash value (which equals the exponentiation required, see Appendix B) in every iteration while adding the value of the nucleotide. Algorithm 2 shows the functioning of this algorithm.

Algorithm 2 *k*-mer computing algorithm with predication and basic arithmetic

```

1: procedure COMPUTE_KMER(String kmer)
2:    $h \leftarrow 0$ 
3:   for char  $c_i$  in kmer do
4:      $h \leftarrow h \ll 2$ 
5:     if nucleotide == C then  $h \leftarrow h + 1$  end if
6:     if nucleotide == G then  $h \leftarrow h + 2$  end if
7:     if nucleotide == T then  $h \leftarrow h + 3$  end if
8:   end for
   return  $h$ 
9: end procedure

```

When compared to the previous one, much lesser instruction dependency occurs in this implementation, but at the expense of predication. Notice also that still a single instruction dependency occurs between $h \leftarrow h \ll 2$ and the additions. Lastly, the third method, namely

Table 5.2 Kernel properties depending on the computation method used. From left to right (1) Occupancy %, (2) Runtime in μs , (3) Predication efficiency %, (4) Average Instruction Per Cycle.

Method	Occ. %	T (μs)	Pred. %	IPC
Predication, binary	93.0%	271	95.2%	2.96
Predication, constants	83.9%	231	64.8%	4.03
Arithmetic	97.7%	366	100.0%	3.25

using cached constants along with predication, can be used to remove any instruction dependency at the expense of the cache. The hash calculating kernel is shown in Algorithm 3.

Algorithm 3 Constant-based k -mer computing algorithm

```

1: procedure COMPUTE_KMER(String  $k$ -mer)
2:    $h \leftarrow 0$ 
3:   for  $k$  in 1 to size( $k$ -mer) do
4:     if  $k\text{-mer}_k == C$  then  $h \leftarrow h + pow_k^{4C}$  end if
5:     if  $k\text{-mer}_k == G$  then  $h \leftarrow h + pow_k^{4G}$  end if
6:     if  $k\text{-mer}_k == T$  then  $h \leftarrow h + pow_k^{4T}$  end if
7:   end for
8:   return  $h$ 
9: end procedure

```

The constant memory (although located in the global space memory) has a dedicated, on-chip cache that enables to broadcast constant values to threads. This is the case of Algorithm 3, since only a small number of different values (namely 4, one for each nucleotide) can be requested from constant memory at the same time. Only four constants can be requested because of warps advancing in lockstep of 32 threads, and thus all threads request one of four constants at any k step in the main loop.

Table 5.2 shows a comparison between the three possible implementations. The predication method accompanied by the fetching of constants achieves the highest IPC. This is due to all threads accessing a maximum of 4 different constants at once, *i.e.* the on-chip constant memory will broadcast the 4 requested values to all threads which, although serialized, are still faster than the instruction dependency caused by the integer operations in both the binary and especially in the arithmetic kernel.

Therefore the complete kernel that computes the hashes of words while also checking for malformed k -mers is depicted in Algorithm 4. Note that ti_x stands for the x component of the *threadIdx* built-in variable that represents threads within a block. In the same line, bi_x is the x component of the *blockIdx* variable, and finally bd_x is the x component of the *blockDim*

variable. Shared memory is used to cache the bytes corresponding to the sequence that will be processed (*i.e.* the number of threads plus the size of the k -mer) to avoid the one-byte shift per thread that causes memory transactions to be split in two parts due to misalignment. Doing so requires a `syncthreads` call in order to prevent threads that are not involved in the caching to advance outside the current lockstep.

Algorithm 4 Words calculation kernel

```

1: procedure COMPUTE_WORDS(Sequence s, Keys keys, Values val)
2:    $h \leftarrow 0$ 
3:    $bad \leftarrow 0$ 
4:   shared scached( $block_{size} + 32$ )
5:    $scached_{ti_x} \leftarrow s_{ti_x + bi_x * bd_x}$ 
6:   if  $ti_x < 32$  then  $scached_{ti_x + 64} \leftarrow s_{64 + ti_x + bi_x * bd_x}$  end if
7:   syncthreads()
8:   for  $k$  in 1 to  $size(k\text{-mer})$  do
9:      $c = scached_{ti_x + k}$ 
10:     $nucleotide \leftarrow s[ti_x + k + bi_x * bd_x]$ 
11:    if  $k\text{-mer}_k == C$  then  $h \leftarrow h + pow_k^{4C}$  end if
12:    if  $k\text{-mer}_k == G$  then  $h \leftarrow h + pow_k^{4G}$  end if
13:    if  $k\text{-mer}_k == T$  then  $h \leftarrow h + pow_k^{4T}$  end if
14:    if  $k\text{-mer}_k == N$  then  $bad \leftarrow 1$  end if
15:  end for
16:   $keys_{ti_x + bi_x * bd_x} \leftarrow h | bad$ 
17:   $val_{ti_x + bi_x * bd_x} \leftarrow (ti_x + bi_x * bd_x) | bad$ 
18: end procedure

```

The storing of the hash (key) and the value (position) in global memory does not require caching and results in a fully coalescent transaction. In fact, since the k iterator is not included in the indexing of the key and value storing, the indexing formula does not allow any misaligned transactions (as long as the original boundaries are aligned). As long as the block size is a multiple of 4, each store operation can be coalesced into one transaction for every warp.

Reverse-complement kernel

Calculating the reverse and complementary strand of the reference sequence must be done only once prior to the computation of words. Comparing the query against the reference is not enough to assess the similarity between two sequences because DNA inversions can take place (which effectively flips the DNA from one strand to the other, analogous to reversing

and complementing the original DNA sequence) and thus the reverse complement must also be compared against [105].

The CPU counterpart implementation of this procedure is inherently fast: a sequential load and complement of nucleotide followed by a buffered write to memory. This means that CPUs will excel at this type of processing, for caches are particularly efficient when processing contiguous data, and this can be seen in the previous profiling analysis, which showed that on average the reverse complement operations requires no more than 1.8% of the processing time. Moreover, the amount of work performed per unit of information is significantly low, in fact requiring only the complementing operation. This reasoning is clearly against a GPU kernel implementation, especially if the size of chromosome sequences are factored in along with the host to device transfers.

However, by employing the following techniques it is possible to overcome the transfer penalty and also produce additional speedup (while also freeing the host from performing the work). In particular:

- Full coalescence in both reads and writes from global memory.
- Use of pinned memory to speed up memory transfers between the host and the device.
- Use of the CUDA asynchronous memory copy operations in order to process the data while it is loaded.
- Running the kernel in lockstep except for the main conditional which only serializes execution in the last block and thus its effect is insignificant.

The corresponding kernel is shown in Algorithm 5. Notice that as opposed to the words kernel, no k parameter intervenes in the execution (*i.e.* every nucleotide is used only once, and not k times) and thus no “walking pattern” prevents coalescence from taking place without buffered shared memory.

Before Algorithm 5 starts, the reference sequence is divided into equally-sized batches and transferred to the device using as many streams as batches; thus processing can begin as soon as partial sections of the reference sequence arrive at the device. Overall, these optimizations yield an approximate 5x speedup over the sequential host version (including the time to transfer from and to the device). Although it is not a massive speedup, this approach not only reduces the runtime but also frees the host CPU to perform any other calculations meanwhile.

Algorithm 5 Reverse complement kernel

```

1: procedure REVCOMP(Sequence  $s$ , Sequence  $r$ )
2:    $id \leftarrow ti_x + bi_x * bd_x$ 
3:   if  $id < |s|$  then
4:     nucleotide  $\leftarrow s[id]$ 
5:     if nucleotide == A then nucleotide  $\leftarrow T$  end if
6:     if nucleotide == C then nucleotide  $\leftarrow G$  end if
7:     if nucleotide == G then nucleotide  $\leftarrow C$  end if
8:     if nucleotide == T then nucleotide  $\leftarrow A$  end if
9:      $r[(|s| - 1) - id] \leftarrow \text{nucleotide}$ 
10:  end if
11: end procedure

```

Word-sorting kernel

The sorting stages are crucial for the GECKO algorithm. In fact, it is the sorting that replaces the need for an indexing data structure (see Section 2.3.3) and that enables the generation of the seeds without requiring binary search on each key. Two sorting stages are executed per comparison, first for words (which then enable to create the seeds) and secondly for seeds (which enables to filter unwanted seeds and prune the search space). The use of sorting mechanisms in the processing pipeline is of great benefit in terms of GPU acceleration, since sorting algorithms are one of the most well-studied algorithms for any computational architecture, as well as the foundation of so many other algorithms [106].

However, there are several particularities that must be addressed in order to choose and to adjust a sorting algorithm that complies with the requirements. In particular, the sorting method must be able to (1) work efficiently with millions of elements per batch and (2) handle long numbers without loss of efficiency. In the case of (1), the typical sequences being compared with GPUGECKO (such as mammalian chromosomes) will require sorting from few to over 10^8 key-value pairs. This broad range of seeds can be seen in the $Seeds = f(W)$ column of Table 5.3, which, if applying the subsequence batching procedure (as explained in Section 5.1.3) on a 4 GB device, yields an average number of 18,407,027 (or $1.8 * 10^7$) seeds along 264 subdivisions.

In the case of a device with more memory available (*e.g.* 8 GB) the number of seeds can be much higher, and thus a suitable sorting algorithm is required. Regarding (2), each pair to be sorted by its key is in the numerical range $[0, 2^{64}]$ in the case of the words sorting, whereas for seed sorting the range is slightly more complex, although still represented by 64 bits. The most commonly used GPU sorting algorithms are quick sort, merge sort and radix sort [107]. On one hand, quick sort and merge sort have the same average complexity

Table 5.3 Number of seeds spawned at word size $k = 32$ for a set of sequences including bacteria and different classes of the Chordata taxonomic class.

Sequences	Length X (bp)	Length Y (bp)	$Seeds = f(W)$
<i>Mycoplasma h.</i> 232, 7422	892,725	898,121	417,587
<i>E. coli B</i> , K12	4,478,925	4,558,627	3,693,670
<i>Gallus g.</i> 18, <i>Meleagris g.</i> 20	10,494,880	9,898,897	604,929
<i>Oryzias l.</i> 6, <i>Danio r.</i> 25	23,318,240	37,418,471	71,410,917
<i>Sus s.</i> 11, <i>Bos t.</i> 12	79,119,489	90,363,562	12,699,274
<i>Homo s.</i> X, <i>Mus m.</i> X	151,099,878	163,484,862	730,598,530
<i>Homo s.</i> 1, <i>Gorilla g.</i> 1	225,279,443	211,487,502	3,136,314,852

$O(n * \log n)$, thus depending only on the number of keys. On the other hand, radix sort has $O(k * n)$ where k is the size of the key (not to be confused with the k -mer word size). Since the keys employed are 64-bit long (both in the sorting of words and seeds), the trade-off between number of items and length of keys is not high enough for the performance of radix sort to pay off [108], since not enough keys will fit into the memory of the device.

Given that the quick sort algorithm has worst time complexity $O(n^2)$ (when sorting nearly and fully sorted datasets) and that it performs worse on average with 64-bits keys than merge sort (see [109]), the merge sort implementation from ModernGPU [110] was employed along with a custom memory allocator to reduce the number of system allocation calls (as well as making use of previously allocated pinned memory). The custom memory allocator replaces any `cudaMalloc` calls that the merge sort algorithm requires by a memory pool which is orchestrated by a stack (since all allocation calls are first performed in order and then in reverse). The code for the custom allocator is available in Appendix C.

Lastly, radix sort also requires auxiliary space, typically $2n$ space in total and especially when using stable sorting which is necessary in the seed sorting stage. Modifications of the radix sort algorithm have been devised to require only constant auxiliary space, however their application is limited to keys of particular size and do not apply to the proposed method (see [111]). On the contrary, merge sort can perform in-place sort, which is a significant advantage when considering the limiting memory size of GPU devices.

Generation of seeds

In the previous sections, the procedures to hash words and their sorting (as well as the reverse complement kernel for the reference) were introduced. However, the goal of these algorithms culminates with the generation of seeds. Once words are sorted by their numerical value (*e.g.* word ATTT comes before CTGT because nucleotide A has a smaller value than C) they can be matched in linear time by sweeping both word dictionaries (query and reference) while

incrementing the index of each dictionary depending on the equality of the currently visited words. This is, while the current word in the query dictionary has a smaller numerical value than the current word in the reference dictionary, then the query index is incremented and vice versa. If the two words match, then a seed is created. This procedure continues until the indices reach the end of the dictionaries. Recall that since the dictionaries are sorted, no word is visited more than once (with an exception which will be addressed later on). The following sections will discuss various methods to generate seeds followed by another section describing the performance analysis and mechanisms implemented to accelerate the calculation of hits.

A note regarding seed-generation in state-of-the-art methods There is one subtle note in the bioinformatics context which is worth discussing: in most seed-based comparison methods (even GECKO or BLAST) a non-exhaustive approach is used to generate seeds. This means that not all pairs (w_i, w_j) are generated, but instead only those that are either unique or whose frequency is less than a given threshold. In fact, in the case of BLAST, low-complexity regions are removed with the DUST program [112] before running the actual comparison. This is designed to avoid highly repetitive seeds (such as AAAAA or CGCG) which can drain the performance of a non-automatically-balancing data structure (such as tree-based structures) or even worse, cause an overflow in the systems where the algorithm is running. In the end, highly repetitive seeds can result in extremely degraded performance. Therefore, most algorithms limit the number of occurrences of a single word pair to a few hundreds or thousands. While this has a clear impact on accuracy, it affects parts of the signal more than others, and therefore which approach to use depends on the scope of the research being done.

Despite this scenario, the proposed method runs by default on exhaustive mode, *i.e.* calculating all pairs of possible seeds and thus producing optimal results (as optimal as a seed-based approach allows). This is possible due to the seed-generation step running completely within the acceleration device. Nevertheless, the proposed method is also equipped with two additional working modes which are designed to mimic the limiting threshold procedure in other state-of-the-art software. Briefly, the idea behind using these other working modes is to increase speed by sacrificing a controlled amount of signals. In the next sections, the three approaches taken by GPUGECKO to generate seeds are described, including probabilistic, uniform and exhaustive.

Probabilistic generation of seeds The probabilistic method for the generation of seeds is based on the well-known “First come, first served” model. However, in this case, only the first one to come is served. That is, for every pair of words (w_i^x, w_j^y) such that $w_i = w_j$,

only those which satisfy $p_x \ll p'_x$ are accounted for and thus generated. This means that if there are two and three equal words in the query and reference, respectively, only the seeds corresponding to the following pairs are generated: (w_1^x, w_1^y) , (w_1^x, w_2^y) and (w_1^x, w_3^y) . Notice that w_2^x is not matched to any of the words in the reference sequence, and that only 3 out of the exhaustive 6 seeds have been generated. In fact, for n and m equal words, this method reduces the number of seeds from $n * m$ to m .

Uniform generation of seeds Contrary to the previous method, which is probabilistic in the sense that a seed has a probability of being generated or not depending on the positional value of the two words that formed it, the following is a method that still prunes the search space but produces seeds in a uniform approach.

In this line, a mechanism is devised to keep the number of seeds linear and uniformly spaced among all possible seeds, without requiring a threshold nor fine-tuning. Given a word that appears n and m times in each sequence (which in sensitive mode would result in $n * m$ seeds), the square root is taken for both so that the number of seeds is $\sqrt{n * m}$, which keeps the amount of seeds in the same order of magnitude as the number of words. This means that the step in which words are skipped is the square root of the number of appearances of the particular word in each sequence, which enables to select them in an evenly-spaced fashion. For instance, consider the same word being repeated 10 and 20 times, respectively. While the original number of spawned seeds would account for $10 * 20 = 200$, the suggested policy only requires $\sqrt{10} * \sqrt{20} \approx 14$. Overall, the equation of the number of seeds for the fast mode is shown below:

$$s_{w_x, w_y} = \sqrt{n} * \sqrt{m} \quad (5.3)$$

Moreover, by knowing the number of words n and m it is possible to select only those seeds that are evenly-spaced. This is achieved by first counting the number of equal words for a given word value, then applying Equation 5.3 and finally selecting only those words which are n or m words apart. While this approach requires to count the number of words per given word value (thus resulting in each word being visited twice), the speedup is massive given how many seeds get removed (*i.e.* quadratically less seeds) which means less computation is required.

Exhaustive generation of seeds The exhaustive generation of seeds presents several problems which are a direct consequence of the unbounded number of seeds that can be generated. As opposed to the probabilistic or uniform method where how many seeds will

be created is known *a priori*, in the exhaustive approach this is unknown. In particular, the following challenges are addressed:

1. Branch divergence. The fact that it is unknown how many seeds will be spawned per thread can result in long code serializations of conditionals, since different actions must be carried out depending on whether seeds are generated or not.
2. Unpredictable load. A static partitioning of the dictionaries might result in extreme load unbalance, *i.e.* some blocks will finish their execution altogether whereas others will idle on particular threads. If partitioning is done dynamically, then synchronization mechanisms are required.
3. Synchronization at grid level. A mechanism to orchestrate the concurrent production of seeds must be devised such that performance is not stopped by store transactions to global memory while balancing block buffers.
4. Synchronization at block level. If threads within a block cooperate to generate seeds together, then they must synchronize to know exactly how many seeds have been generated by any of them in order to not produce gaps between stores.
5. High overhead per small job size. The calculation of each seed requires nearly no computation besides a couple of arithmetic operations. This incurs high overhead if synchronization mechanisms of some sort are used to balance the load.

Regarding (1), most of branch divergence can be avoided by reducing conditionals to the maximum, *i.e.* allow for a small amount of duplicate work (*e.g.* calculate a diagonal that will not be stored) as long as the only conditionals left are replaced by predication. By carefully orchestrating the use of shared memory, most of the overhead in (5) can be compensated for with higher memory bandwidth. Regarding load balancing (2), a function of the type $\frac{1}{x}$ is used to produce smaller blocks of work as processing progresses (similar to the function used in Equation 4.2). This function encodes increasingly smaller ranges of sorted words (indices) to minimize threads from blocking the processors in which they are executing while other threads are idling.

The last two problems concerning synchronization (3 and 4) require the incorporation of both algorithmic changes and application-domain information.

Regarding the synchronization of stores, two mechanisms are devised, one at block level to deal with inter-thread writes and one at grid level to deal with inter-block writes. These two mechanisms are also coupled with two memory spaces; one space where each block has a reserved section, and an additional set of larger-but-unowned sections. The key idea is that

each block will write an approximately equal amount of seeds to their own dedicated space. However, the blocks that will generate an almost quadratic amount of seeds (see section 2.3.2 "Number of patterns") can claim additional spaces by incrementing an atomic counter. This does not represent a drain of performance since these atomic calls occur randomly and rarely (in the order of few hundred times per comparison), and thus blocks have an extremely small chance to block each other while requesting an owned memory space by making use of the atomic counter. This conforms the first mechanism that deals with stores at the grid level.

The second mechanism, which deals with stores at the thread level, is concerned with how threads within a warp synchronize to write the produced seeds to the dedicated memory space. It is possible to take advantage of the fact that words are sorted (see Section 5.1.2): if no seed is generated by thread t_i , then it is known that t_{i+j} will not generate a seed either. This effectively translates into only requiring two indices to pinpoint the range of threads $t_s, t_{s+1}, \dots, t_i$ that have generated seeds, *i.e.* the index of the first thread that has produced a seed, and the number of threads that have produced seeds in total. These two indices are distributed using min and sum shuffle reduction respectively. Note that if words were not sorted, it would be impossible to know whether a thread in-between t_s and t_i has produced a seed unless distributing one value per each thread. However doing so would render the algorithm useless since it would require more compute time to distribute all these indices (as many as threads in a warp) instead of actually computing word matches into seeds.

Altogether, the exhaustive seed-generation kernel (depicted in more detail in Algorithm 6) follows the next four steps:

1. First, every block gets assigned a batch of sorted words from the query dictionary and performs binary search on the reference dictionary to find out the first reference key that could match against the query.
2. Shared memory is used to greedily cache the query keys along with the values (positional information) before they are actually needed. This pays-off since it allows for full coalescence (and it is known due to the binary search that such keys will be processed).
3. These keys are then scanned against consecutive reference keys starting from the binary search index and write the results to the appropriate memory space using the synchronization mechanism explained above. Depending on the equality of the last matched keys, the scanning continues or a new set of query keys is cached.
4. When all query keys are exhausted, a compaction step is carried out to remove gaps between the inter-block writes. This allows to save both storage in global memory and reduce the size of the last sorting procedure.

Algorithm 6 Pseudocode of the seed-generation kernel

```

1: procedure GENERATE_SEEDS(Words  $w_{xy}$ , Array  $m$ , Atomic lock)
2:   shared keysxy[32]
3:   startx, endx  $\leftarrow$  ranges( $b_x$ ,  $g_x$ )
4:   starty  $\leftarrow$  bsearch(startx,  $w_y$ )
5:   while startx < endx do
6:     keysx  $\leftarrow$   $w_x$ [startx +  $t_x$ ]
7:     do
8:       keysy  $\leftarrow$   $w_y$ [starty]
9:       if keysx[startx] == keysy[ $t_x$ ] then  $t_s \leftarrow t_x$ , hit  $\leftarrow$  1 end if
10:       $t_s \leftarrow$  shuffle_reduction( $t_s$ , min())
11:       $m[t_s + t_x] \leftarrow$  diagonal(keysx, keysy)
12:      hit  $\leftarrow$  shuffle_reduction(hit, sum())
13:      accumulated  $\leftarrow$  accumulated + hit
14:      if accumulated >  $m_{size}$  then
15:         $m \leftarrow$  atomic_update( $m$ , lock)
16:      end if
17:      starty  $\leftarrow$  starty + 32
18:      while keysx[startx]  $\leq$  keysy[0]
19:        startx  $\leftarrow$  startx + 1
20:      end while
21: end procedure

```

These procedures and mechanisms result in a kernel that is able to achieve full coalescence (both in global and shared memory) while dealing with all the challenges described in 5.1.6. The performance of the kernel is then limited only by the execution dependency caused by the warp shuffle instructions used within the intra-block synchronization mechanism.

Sorting of seeds

The sorting of seeds proceeds analogously to the sorting of words (see section 5.1.6 "Word-sorting kernel" except for one difference: while in the case of words only one field (namely the numerical hash) was used as key for the sorting, in the case of seeds the sorting has to be done on two keys. These two keys correspond to the diagonal value of the seed and the position in one of the two sequences. Sorting by these two keys is fundamental to enable proper filtering of seeds. However, state-of-the-art sorting algorithms for GPU devices are not designed for ordering data based on multiple keys, but rather focus on two scenarios: either only one key and value [107] or one key and multiple values [113]. Multiple-key sorts typically use concatenation of the distinct key fields and then perform a sort; nonetheless this approach usually results in long keys where memory bandwidth starts to become an even

more limiting factor in terms of performance. Therefore, to avoid such loss of performance, a merging (as opposed to simple concatenation) approach is employed based on numeric boundaries on the two keys such that both fit in a single 64-bit key. The advantages of this method are several:

- By using a standardized key size, all GPU sorting algorithms can be employed, regardless of limitations that depend on the number and size of keys and fields.
- The merging of the two keys can be done in constant time at the time of seed generation, *i.e.* nearly no overhead is added.
- The sorting algorithm does not require additional comparisons per original comparison, neither is memory affected negatively by higher bandwidth requirements.

In short, the method consists of merging the diagonal and positional value into a single 64-bit key. For simplicity, assume that sequences s_x, s_y have equal length and thus $|s_x| = |s_y| = n$.

The merging of the diagonal d and the positional values s_x, s_y into one key is possible because of the ternary relationship $d = s_x - s_y$. Equation 5.4 shows the calculation of the merged key.

$$d_{merge} = d_{prev} * d_{length} + p_{query} \quad (5.4)$$

Where d_{prev} is the raw diagonal value calculated as $d_{prev} = p_{query} - p_{reference}$ with d_{length} being the maximum length of diagonal (calculated as $d_{length} = \max(q_{length}, r_{length})$), p_{query} and $p_{reference}$ being the seed positions in both query and reference and lastly q_{length}, r_{length} being the lengths of the input sequences. The key idea is to partition the 64-bit key into consecutive segments of the diagonal size. This means that the diagonal value imposes the first level ordering, and the position along the query coordinate is the second level. Note that no overlap between diagonals can take place since the query position cannot be larger than the length of the diagonal, by definition.

Notice that since Equation 5.4 effectively compresses two keys which are dependent on length into a single one, it is theoretically possible to input long enough sequences that overflow the merge approach. The upper bound for the sizes of the input sequences is calculated as follows (assume $q_{length} = r_{length}$):

$$\begin{aligned}
2^{64} &= n_{diagonals} * d_{length} \\
2^{64} &= (q_{length} + r_{length}) * \max q_{length}, r_{length} \\
2^{64} &= 2 * q_{length} * q_{length} \\
2^{64} &= 2 * q_{length}^2 \\
\sqrt{\frac{2^{64}}{2}} &= q_{length}
\end{aligned} \tag{5.5}$$

This results in an operational range of $2,147,483,648 * \sqrt{2} \approx 3 * 10^9$, or approximately 3 Giga base pairs. This length is sufficient for any sequence comparison due to the following reasons:

- The upper bound in length is not for the input sequences per se, but rather for the local coordinates used in each of the subsequences. Thus, for this number to become a limiting factor it would require a GPU device with enough memory to hold subsequences of 3 Gbps. This is especially unlikely since subsequences of 3 Gbps would require up to 480 Gigabytes of global memory (see 5.1).
- Secondly, such length is superior to any biologically reasonable length, *i.e.* the length of chromosomes has an upper bound [114], typically a few hundred megabase pairs. Thus, even in the hypothetical case where such length could become a problem, it would still be feasible to simply separate input sequences by chromosomes and run them individually.

Filtering of seeds

Sorting seeds by their diagonal value and their query or reference position will order the seed dictionary in such a way that adjacent seeds are consecutive, and therefore, can be removed in a posterior filtering step. This is done to prevent duplicates of HSPs, since overlapping seeds produce the same seed extension. Notice that if the query or reference position is not included in the sorting, then seeds will be ordered by diagonal but disordered per each diagonal, which prevents filtering.

The kernel is depicted in Algorithm 7 and begins with the set of merged and sorted diagonals. First, each thread unpacks the merged diagonal into the original diagonal d , and the $w_j^{p_x}$ and $w_j^{p_y}$ values, corresponding to the coordinates of the seed. The second part of the kernel verifies which seeds must be filtered out, and it does so by iteratively exchanging d_i and $w_i^{p_x}$ with threads t_j such that $j > i$. This means that for words of size k , $k - 1$ exchanges

and checks are performed. These could be done linearly by only one thread, however the same exchanges would be required. Note also that running such a filter in parallel implies that overlapping seeds cannot be checked (and therefore discarded) within the boundaries of each block. This results in few seeds not being filtered out. In the case of the proposed kernel, since each block computes in sections of 32 seeds (to match warp size), there is a maximum slippage of one unfiltered seed for every 32.

Algorithm 7 Seed-filtering kernel

```

1: procedure FILTER_SEEDS_KERNEL(Diagonals  $d_{merged}$ , Integer  $reflen$ , Integer  $n_{hits}$  )
2:    $id \leftarrow blockIdx_x * blockDim_x + threadIdx_x$ 
3:    $v \leftarrow d_{merged}[id]$ 
4:    $d_{curr} \leftarrow (v \& mask_d) >> 32$ 
5:    $px \leftarrow (v \& mask_{px})$ 
6:    $py \leftarrow reflen + px - d_{curr}$ 
7:    $unpacked \leftarrow (px << 32) + py$ 
8:   for  $i$  in 0 to  $blockDim_x - 1$  do
9:      $d_{prev} \leftarrow \text{shuffle\_exchange}(d_{curr}, i);$ 
10:     $px_{prev} \leftarrow \text{shuffle\_exchange}(px, i);$ 
11:    if  $threadIdx_x > i$  and  $d_{prev} == d_{curr}$  and  $px < (px_{prev} + 64)$  then
12:       $unpacked, px, d_{curr} \leftarrow 0xFF..FF$ 
13:    end if
14:  end for
15:   $d_{merged}[id] \leftarrow unpacked$ 
16: end procedure

```

Regarding its parallel design, the kernel is fully coalescent (as diagonals are loaded and stored in batches of warp sizes at aligned addresses) and no divergence occurs despite the conditional on line 10; in fact this conditional is predicated due to its simplicity. The performance of the kernel is limited by execution dependency rather than memory, since the latter is only used for the coalescent fetch of diagonals, whereas warp-shuffle instructions are used for synchronization. Although this results in increased execution dependency, it is faster than using shared or global memory to distribute the diagonal and positional values. In terms of occupancy, the kernel can be launched with a minimum thread block size equal to that of the warp size. Nevertheless, higher performance can be achieved by increasing the number of threads by multiples of the warp size. In the particular case of the GeForce 980 GTX, 64 threads are used (double the size of the warp) to increase occupancy to nearly 90%, whereas using warp-sized blocks results in only half the occupancy. While this does not translate to a directly proportional increase in performance, in most cases it can increase efficiency. Note that increasing the number of threads per block must be taken into account due to the

warp shuffle instructions, in order to avoid attempting to synchronize lanes corresponding to threads from different warps.

Extension of seeds

The seed-extending kernel is the most unpredictable one in terms of heterogeneous workload along with the seed-generation kernel. This is a result of the arbitrary size of the final alignments. An alignment can range easily between less than 50 base pairs to several thousands (see Figure 2.4 "Distribution of High-scoring Segment Pairs by length"). Whether one seed will spawn a short or long fragment cannot be known until actual computation, and thus this must be taken into account when designing the CUDA kernel. Particularly, the most performance-draining drawback that needs to be avoided is branch divergence and ultimately blocks of threads being kept in an SM while most of the warps are idling. This behaviour is related to the grain size of the kernel, namely whether alignments are calculated on a per-thread or per-block basis. In particular:

- In a per-thread fashion, each thread computes one seed-extension.
 - This enables higher occupancy, *i.e.* the kernel can be run with any block size.
 - The kernel benefits from latency hiding since while the memory requests await to be serviced or the pipeline is blocked on an instruction dependency, another thread can be serviced or computed.
 - Each thread calculates a different step, therefore no “redundant” computation is performed to keep warp lockstep.
 - The memory bandwidth is the bottleneck since every thread requires a different transaction.
- In a per-block fashion, each block runs a warp of 32 or more threads on a single seed-extension.
 - This avoids a block being held onto a SM and locking up resources, since seeds can extend into arbitrary long HSPs.
 - No branch divergence. In a per-thread fashion, there is the possibility of high divergence, which ultimately gets serialized. In a per-block fashion, threads share the calculation of a single seed, and therefore they finish at the same time.
 - Requires synchronization between the threads in a block and therefore the bottleneck can switch from memory to compute.

Each implementation design has several benefits and disadvantages. Nevertheless, empirical experimentation showed that in the circumstances of DNA comparison, the per-block fashion was able to achieve higher acceleration, up to a factor of 5 times.

In the per-block kernel, each block runs in complete lockstep and achieves nearly optimal coalescence (with the exception of the walking pattern), aligning contiguous 32-byte sections of the sequences in a collaborating fashion between threads. The hotspot of the kernel is the sum reduction procedure performed using warp shuffle intrinsics, as shown in Algorithm 8. Note also that for each subsequence, the seed-extending kernel first computes the seed-to-5' direction (forward) and then the seed-to-3' direction (reverse complement).

Algorithm 8 Seed-extending kernel for the seed-to-5' direction

```

1: procedure EXTEND_SEEDS(Seed seed, Sequence s1, Sequence s2)
2:   score  $\leftarrow$  32
3:   while score > 0 do
4:      $x \leftarrow seed_x - 32 + threadIdx_x$ 
5:      $y \leftarrow seed_y - 32 + threadIdx_x$ 
6:     similarity  $\leftarrow s_{1,x} == s_{2,y}$ 
7:     for offset = 16; offset > 0; offset  $\div$  2; do
8:       similarity  $\leftarrow$  similarity + shuffle_exchange(similarity, offset);
9:     end for
10:    similarity  $\leftarrow$  shuffle_sync(similarity, 0)
11:    score  $\leftarrow$  score + similarity - (32 - similarity)
12:    seed  $\leftarrow$  seed - 32
13:   end while
14: end procedure

```

The working principle of the kernel is simple: since a coarse-grained approach (per-thread fashion) results in too much unpredictability, let the threads collaborate in a fine-grained approach by calculating each partial result, then aggregating it and lastly broadcasting it using warp intrinsics. In particular:

1. Initially the score is set to the length of the seed (32 points).
2. Each thread starts at a different position in the current 32 nucleotide section by an offset equal to the thread ID (thus one per position from 0 to 31 included).
3. While the score is positive:
 - (a) Each thread computes whether the nucleotide corresponding to its position in both the query and reference sequence are equal, assigning either 1 or 0 to this operation in the similarity variable.

- (b) A sum and reduction is performed on the calculated similarity variable among pairs of threads iteratively until thread t_0 holds the sum. This is done in four iterations through the `shfl_down_sync` primitive by summing t_0 with t_{16} , t_1 with t_{17} , ..., and t_{15} with t_{31} in the first iteration (offset of 16 lanes). In the second iteration t_0 is added with t_8 , t_1 with t_9 , etc, and so on (offset of 8 lanes). Eventually the full summation resides in thread t_0 when the offset is 0.
 - (c) The sum is broadcasted to all the other threads from t_0 .
 - (d) Now that every thread knows how similar the current contiguous block is, add or subtract (depending on the direction of the alignment) the next 32 bases and go to Step 3.
4. If the score is negative, stop the alignment and report the position where the highest score was attained.

Note that this procedure enables all threads to work collaboratively, splitting the sequential work evenly to all threads in a parallel fashion while achieving the maximum coalescence possible (due to the walking pattern) at the same time. Regarding the performance of the kernel, the limiting factor is the number of arithmetic operations that have to be performed, particularly the several adds, subtractions, multiplications and divisions which are due to calculating the current percentage of identity, keeping the highest possible score, *etc.*

A further optimization is required for the seed-extension step. Due to the nature of DNA, a significant portion of the seeds might be spaced along the very same diagonal but at a distance larger than the maximum filtering threshold (more than twice the length of a seed). Hence, when seeds are extended in a parallel fashion (*i.e.* one per block of 32 threads in this case) there is a risk that work is repeated amid blocks. This is due to one seed being extended into an HSP that fully overlaps another seed that is at a longer distance than twice the seed size. Unless blocks are synchronized in some way, threads cannot tell if a previous block has effectively extended (or is still extending) a seed that now covers the new seed to be extended. This can become a serious performance drain if two sequences are very similar and thus share many seeds that result in long HSPs (resulting in the same HSP being calculated over and over). Table 5.4 shows a sample output of the *Escherichia coli B* vs *Escherichia coli K-12* without performing a post-filtering to remove duplicate HSPs. It can be seen that several HSPs are duplicates of the same one but with a slight variation (due to the lockstep extension of 32 bp per iteration, which makes the starting or finishing positions vary by a couple of bases). All these HSPs spawned from seeds with greater distance than 64 bp between them (therefore it was not possible to discard them). Notice how HSPs in Table 5.4 (right) comprise nearly 100% of overlap and are only separated by a small distance.

Table 5.4 Sample output of HSPs that have been computed in parallel causing them to overlap one with each other without a synchronizing mechanism.

Begin _x	Begin _y	End _x	End _y	Strand	Length
59,350	4,179,531	61,398	4,181,579	f	2,048
59,355	4,179,536	61,403	4,181,584	f	2,048
59,366	4,179,547	61,382	4,181,563	f	2,016
59,365	4,179,546	61,381	4,181,562	f	2,016
59,364	4,179,545	61,380	4,181,561	f	2,016
59,363	4,179,544	61,379	4,181,560	f	2,016
59,361	4,179,542	61,409	4,181,590	f	2,048
59,360	4,179,541	61,408	4,181,589	f	2,048

While these could be easily filtered out afterwards, calculating these redundant alignments incurs in slowdown. Therefore a static partitioning is introduced to avoid the computation of these duplicates.

The unaware recomputation of seeds can reduce seed-extension performance from nearly 60x to only 4x (compared to the sequential CPU). To overcome this, a static partitioning is performed on the seeds. Instead of calculating one seed per block, an additional external loop is added to force one block to calculate several adjacent seeds. This enables each block to calculate consecutive seeds and check whether the last extended seed encloses the new one before extending it. This probabilistic approach removes a high percentage of the repeated HSPs that get calculated several times, with the probability of removing them increasing with the number of seeds assigned to each processor. Notice that if for instance the partitioning is performed with two seeds per block, then on average only a duplicated HSP can be avoided. On the other hand, if a large number of seeds is used (*e.g.* 50) then a lot more can be removed, with the best case nearing 50 and the average case in the middle, *i.e.* 25 since it is not guaranteed that all expendable and consecutive seeds fall in the same static partitioning. In this case, a shared-memory mechanism was included to store the positions of the seeds to be calculated (so only one global load is required per block) and their extensions, which then get stored into global memory again at once by all threads in a block at once. The seeds per block were set at 32 in order to fit the mechanism to the number of threads that process each seed, and also to balance the amount of shared memory, registers and thus occupancy.

The static partitioning is implemented as an outer loop in the seed-extension kernel, so that the block b_0 takes seeds s_0 to s_k , block b_1 takes seeds s_k to s_{2k} , *etc.* This is achieved by simply assigning consecutive seeds to each block and buffering the results of each seed in shared memory in order to also increase the transaction efficiency of the write operations.

Moreover, this also makes the transactions coalescent by grouping them together at an aligned memory region. Such implementation is described in Algorithm 9.

Algorithm 9 Static partitioning for seed arrays

```

1: procedure STATIC_PARTITION(Seeds  $seeds$ , Sequence  $s_1$ , Sequence  $s_2$ , Seeds
    $extended_{seeds}$ )
2:    $shared$   $s[32]$ ,  $ext[32]$ ,  $diag[32]$ 
3:    $prev_d \leftarrow null$ ,  $prev_{ext} \leftarrow null$ 
4:    $s_{x,y}[thread_{id}] \leftarrow seeds_{x,y}^{id}$ 
5:    $diag[thread_{id}] \leftarrow seeds_x^{id} - seeds_y^{id}$ 
6:   for  $frag$  in  $1..32$  do
7:     if  $prev_d == diag[frag]$  and  $overlap(prev_{ext}, s_x)$  then
8:        $ext_{x,y} \leftarrow 0$ 
9:     else
10:       $ext_{x,y}[frag] \leftarrow left\_extend(s_x, s_y)$ 
11:       $ext_{x',y'}[frag] \leftarrow right\_extend(s_{x'}, s_{y'})$ 
12:       $prev_d \leftarrow diag[frag + 1]$ 
13:       $prev_{ext} \leftarrow ext_{x,x'}[frag]$ 
14:    end if
15:     $n_{frag} \leftarrow n_{frag} + 1$ 
16:  end for
17:   $extended_{seeds}[thread_{id}] \leftarrow ext_{x,y}[thread_{id}]$ 
18: end procedure

```

Note that the whole kernel works in lockstep and that no thread can diverge due to the only conditional being a block-wide one. In fact, it is in charge of checking if the current seed should be extended or not based on whether the previous seed is both on the same diagonal and also extends up to the point of overlapping. If such is the case, then the seed can be skipped without losing any signal, and it does so for the whole warp of the block. If not, then the seed is extended as described with Algorithm 8. Lastly, shared memory is used to buffer (1) the seeds to be extended, which are loaded in coalescence in line 5, (2) the extensions per each seed, in order to coalesce the writes and (3) the diagonal value of each seed is calculated in parallel (one diagonal per thread) as opposed to all calculating it every time. This last stage also benefits from the broadcast capability of the shared memory since all threads access the same diagonal value.

Regarding a dynamic partitioning approach, it would require additional mapping of seeds to blocks (such as tables or lists) which in turn require higher memory usage, as well as a CPU sweep to actually map them. On the contrary, the static partitioning is extremely efficient and simple to implement.

5.2 Results

5.2.1 Introduction

In the following sections, the proposed GPU-accelerated method for the irregular alignment of sequences is tested. General experiments concerning speedup in regards to the CPU counterpart as well as tests targeting a specific aspect of the proposed method are also included. In particular:

1. Runtime comparison with the CPU counterpart.
2. State-of-the-art comparison.
3. Comparison using different Graphics Processing Units.
4. Complexity analysis.
5. Large-scale genome comparison study.

5.2.2 Infrastructure

The following setup including four different graphic card types was employed to run the experiments described in advance:

1. A 980 GTX configuration including 2x Intel Xeon E5-2698 v3 processor at 2.3 GHz, 256 GB of RAM and 4x NVIDIA GeForce GTX 980 with 4 GB of GDDR5 RAM.
2. A 1080 Ti configuration including 1x Intel Xeon E5-2609v4 at 1.7 GHz, 32 GB of RAM and 2x NVIDIA GeForce GTX 1080 Ti with 11 GB of GDDR5X RAM.
3. A 2080 Ti configuration including 1x Intel Xeon E5-2630V4 at 2.2 GHz, 64 GB of RAM and 2x NVIDIA GeForce RTX 2080 Ti with 11 GB of GDDR6 RAM.
4. A 3090 RTX configuration including 1x Intel Xeon E5-2609v4 at 1.7 GHz, 64 GB of RAM and 1x NVIDIA GeForce RTX 3090 with 24 GB of GDDR6X RAM.

Unless stated otherwise, the default configuration used was the 980 GTX setup.

5.2.3 Datasets

The following datasets were used in the benchmarks and comparisons:

1. Single pairs of bacteria and chromosome sequences were used in increasing size, namely *Mycoplasma hyopneumoniae* Chr. 232, *Mycoplasma hyopneumoniae* Chr. 7422, *Escherichia coli* Chr. B, *Escherichia coli* Chr. K12, *Gallus gallus* Chr. 18, *Meleagris gallopavo* Chr. 20, *Oryzias latipes* Chr. 6, *Danio rerio* Chr. 25, *Sus scrofa* Chr. 11, *Bos taurus* Chr. 12, *Homo sapiens* Chr. X, *Mus musculus* Chr. X, *Homo sapiens* Chr. 1, *Gorilla gorilla* Chr. 1.
2. Two collections of chromosomes regarding the *Homo sapiens* and *Mus musculus* genomes.

5.2.4 Runtime comparison with the CPU algorithm

The first experiment is intended to demonstrate the overall acceleration of each of the described kernels in section 5.1 "Methods". In order to make the comparison as fair as possible, the reported speedup acceleration values for the GPU algorithm were calculated including both the running and the host-to-device transfer times. The CPU counterpart was executed using only one core. Still, it should be noted that a significant part of the speedup observed between the sequential GECKO and the proposed GPU implementation is mostly due to (1) the CPU version requires to offload large data structures to secondary memory, whereas GPUGECKO leverages the GPU compute capability by trading off storage space for time using the subsequence batching strategy and (2) GECKO could have been furtherly optimized by *e.g.* vectorizing hotspots.

The averaged runtime and speedup can be seen in Table 5.5 and Table 5.6 for each of the sample datasets (*Hyopneumoniae* 232-7422, *Escherichia coli* B12-K, chromosomes 18 and 20 from the *Gallus gallus* and the *Meleagris gallopavo* genomes and lastly chromosomes 6 and 25 from *Oryzias latipes* and *Danio rerio*). Note that these datasets were selected such that they all fit into the global memory of the GPUs and thus simplify the timing process since no subsequence batching is required. Additional experiments measuring the full runtime for sequences which required subsequence batching are available in subsection 5.2.5 "State of the art comparison".

1. The reverse complement kernel achieves an average of 7x in regards to the CPU version. This is expected (as explained in section 5.1.6 "Reverse-complement kernel") since

Table 5.5 Speedup comparison between GECKO and the proposed implementation GPUGECKO for each kernel, part one. The columns represent (1) CPU time, (2) GPU time and (3) average speedup (μ_s).

Comparison	M. hyopneumoniae 232 - 7442			E. coli B12 - K		
Kernel	CPU	GPU	μ_s	CPU	GPU	μ_s
Reverse Complement	0.087	0.0078	11.17	0.344	0.045	7.59
Words Dictionary	1.162	0.0085	136.76	4.248	0.014	297.30
Words Sorting	0.891	0.0041	216.14	3.668	0.023	157.55
Seeds Generation	1.621	0.0223	72.75	5.989	0.039	154.44
Seeds Sorting	0.187	0.0006	330.05	1.350	0.006	235.85
Seeds Filtering	0.020	0.0008	24.87	0.158	0.008	18.70
Seeds Extension	0.330	0.0019	176.70	0.514	0.021	24.25

Table 5.6 Speedup comparison between GECKO and the proposed implementation GPUGECKO for each kernel, part two. The columns represent (1) CPU time, (2) GPU time and (3) average speedup (μ_s).

Comparison	G. gallus 18 - M. gallo. 20			O. latipes 6 - D. rerio 25		
Kernel	CPU	GPU	μ_s	CPU	GPU	μ_s
Reverse Complement	0.728	0.102	7.16	1.914	0.284	6.74
Words Dictionary	8.613	0.025	349.40	22.929	0.082	280.33
Words Sorting	7.889	0.060	132.28	24.159	0.177	136.54
Seeds Generation	8.606	0.057	151.26	24.264	0.409	59.39
Seeds Sorting	0.266	0.001	331.26	22.303	0.123	181.72
Seeds Filtering	0.034	0.001	25.92	2.167	0.158	13.68
Seeds Extension	0.661	0.003	237.80	7.055	0.056	125.04

the amount of work the kernel requires is substantially low (a byte fetch followed by one arithmetic operation and a store). Nevertheless, some speedup is achieved by making use of asynchronous memory transfers, which not only is desirable but also frees the CPU from performing the work by itself. Also note that the trend appears to be a decrease in speedup in respect to sequence size; however this value stabilizes at a speedup of 5x for longer sequences (*e.g.* 5x and 5.4x for the larger *Hominidae* comparisons).

2. The word dictionary kernel exhibits a clear performance increase relative to the size of the data, *i.e.* the larger the dataset, the higher the acceleration. This is a result of the kernel being purely linear and thus requiring more data to compensate for transfer times.
3. The sorting kernels show some variability between the words sort and the seeds sort (around 200x to 100x difference in speedup). This is due to the seeds being many more in number and thus requiring both larger sorting times and transfer times. Some variation is also observed among dataset sorts due to the number of sorting passes required which depends on the intrinsic similarity of the dataset.
4. The seed generation produces on average between 60x to 150x speedup. The reason behind the high variability is due to the CPU version being purely linear, *i.e.* the number of times a word is visited is strictly one (and the actual bottleneck is the production of the seeds, which can be quadratic in terms of the number of words) whereas in the case of the GPU kernel, the largest acceleration is achieved when there is a uniform and contiguous number of adjacent words together, since this allows to create multiple seeds per warp in one synchronization step. On the contrary, the smaller the number of adjacent and similar words, the higher the number of synchronizations and writes to the memory regions required, as well as the number of interleaved gaps between writes. The described behavior of less adjacent seeds can be seen in Table 5.7 in the form of an extremely low coverage for the ORY-DAN experiment, which is the one that causes the least acceleration.
5. The seed filtering kernel follows a similar example to that of the Reverse Complement one. The linear behavior of the kernel makes it harder for the collaborative approach (which uses shuffle exchange instructions) to achieve high speedup. Still, a decent 13x to 25x acceleration is observed.
6. The seed extension kernel achieves over 100x speedup (and over 200x in some cases) in all datasets except for the *Escherichia coli* one. On one hand, the massive speedup

occurs due to the collaborative and memory-coalescing algorithm which is able to take advantage of the GPU capabilities. On the other hand, the *Escherichia coli* dataset shows less speedup due to the extreme number of repeated seeds that overlap with one another, thus resulting in recomputation. The static partitioning mitigates this effect by increasing the speedup from approximately 4x to 25x. Using a larger number of seeds per block can increase further performance in the *Escherichia coli* dataset, however at the expense of reducing it in the other datasets and producing more duplicate alignments.

The inclusion of larger datasets that do require subsequence batching are studied from the perspective of the full runtime (as opposed to per-kernel), since the added transfer times and the recomputing of subsequences (particularly in the creation of words and sorting) make the performance assessment harder. In the next section, all experiments of the sample dataset are computed and their performance is analyzed.

5.2.5 State of the art comparison

The following experiments comprise the runtime of GECKO and the proposed implementation GPUGECKO. Additionally, the only comparable GPU aligner (which makes use of the seed-and-extend method), namely GBLASTN [115], is shown for comparison purposes. Each of these methods is tested on two approaches depending on the way that seeds are generated (either exhaustively or heuristically, as discussed in section 5.1.6 "Probabilistic generation of seeds" and section 5.1.6 "Exhaustive generation of seeds"). The generated signal is also analyzed in terms of coverage (*i.e.* the ratio of detected alignments above a given threshold of similarity to the length of the sequences involved).

Figure 5.3 shows the runtime for the exhaustive approach. Figure 5.4 shows the runtime for the heuristic approach. In addition to the heuristic mode, the CPU algorithm GECKO was run with four cores.

The first thing to notice is the fact that GBLASTN is not able to complete all of the executions for the larger comparisons (comparison ORY-DAN in exhaustive mode, and comparison HOM-GOR in heuristic mode), whereas GECKO and the accelerated GPUGECKO do finish all executions. The reason behind GBLASTN crashing is the huge number of seeds which need to be stored in-memory given the design of GBLASTN. Regarding computation time, GPUGECKO is initially slower than GBLASTN for the first comparison (due to memory initialization, both in the device and pinned in the host) but quickly becomes faster as the size of the input sequences grows. This is particularly true for the largest comparison, where GPUGECKO is 101 times faster than GECKO and nearly 12 times faster than GBLASTN

in the last available comparison. In summary, GPUGECKO is faster by 1 to 2 orders of magnitude in comparison with GECKO and GBLASTN.

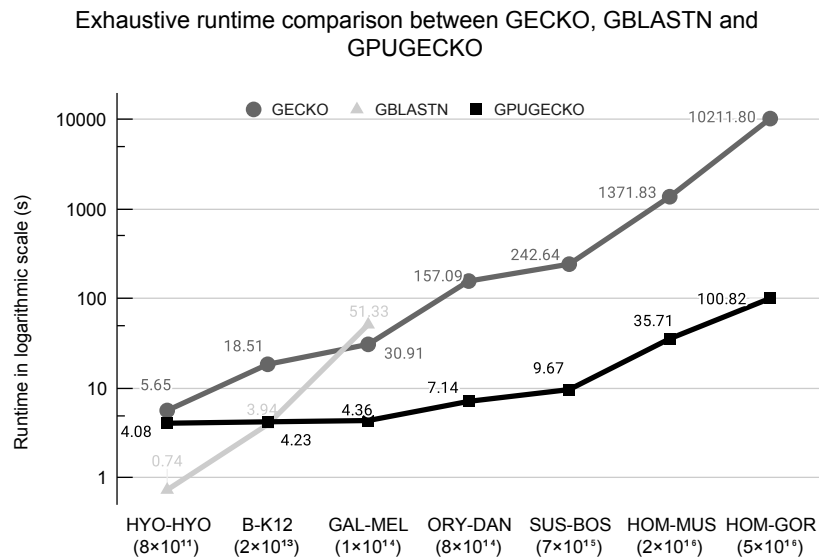


Fig. 5.3 Runtime comparison between the three algorithms in the exhaustive approach. The x-axis iterates through all sample datasets ordered by increasing search space size. The y-axis shows the runtime in logarithmic scale. The GPUGECKO algorithm is represented with black square lines, GECKO is shown with gray circles and GBLASTN with light gray triangles.

Regarding coverage in the exhaustive execution, as can be seen in Table 5.7, GPUGECKO is able to find the most signal in all of the comparisons. This is expected since the algorithm is designed to keep and extend every seed while optimizing the length of each alignment rather than percentage of identity (which is fixed at a threshold of 80%). This means that for the same set of seeds GPUGECKO is likely to report more coverage due to the inclusion of longer alignments of a slightly less similarity. Note that coverage is calculated as the number of aligned base pairs divided by the length of the query sequence.

In the case of the heuristic execution (see Table 5.8), small variations in coverage are observed in comparison with the exhaustive methods, with GPUGECKO achieving higher coverage on all executions except the SUS-BOS one. The reasons behind this variation are simply due to the seed-skipping policies, which optimize the amount of coverage depending on the underlying seed distribution.

Heuristic runtime comparison between GECKO, GBLASTN and GPUGECKO

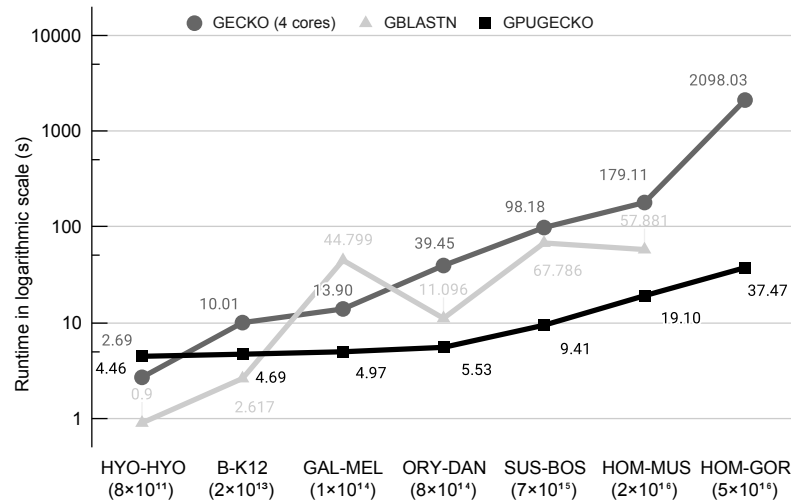


Fig. 5.4 Runtime comparison between the three algorithms in the heuristic approach (and using four parallel cores in the case of GECKO). The x-axis iterates through all sample datasets ordered by increasing search space size. The y-axis shows the runtime in logarithmic scale. Same representation as in the previous Figure applies.

Table 5.7 Runtime and coverage comparison between the three methods for the exhaustive run. Comparisons are ordered by the size of their search space. Time is measured in seconds and coverage is measured in percentage. The acronym "DNF" stands for "Did Not Finish".

Exhaustive run	GECKO		GBLASTN		GPUGECKO	
	T (s)	Cov. (%)	T (s)	Cov. (%)	T (s)	Cov. (%)
HYO-HYO (8×10^{11})	5.65	90.10	0.74	92.80	4.08	94.93
B-K12 (2×10^{13})	18.51	93.14	3.94	91.06	4.23	94.66
GAL-MEL (1×10^{14})	30.91	49.85	51.33	58.53	4.63	59.61
ORY-DAN (8×10^{14})	157.09	0.23	DNF	DNF	7.14	0.25
SUS-BOS (7×10^{15})	242.64	5.59	DNF	DNF	9.67	5.67
HOM-MUS (2×10^{16})	1,371.83	0.99	DNF	DNF	35.71	1.02
HOM-GOR (5×10^{16})	10,211.80	66.31	DNF	DNF	100.82	69.68

Table 5.8 Runtime and coverage comparison between the three methods for the heuristic run. Comparisons are ordered by the size of their search space. Time is measured in seconds and coverage is measured in percentage. The acronym "DNF" stands for "Did Not Finish".

Heuristic run	GECKO		GBLASTN		GPUGECKO	
Comparison	T (s)	Cov. (%)	T (s)	Cov. (%)	T (s)	Cov. (%)
HYO-HYO (8×10^{11})	2.69	90.10	0.90	92.22	3.59	94.91
B-K12 (2×10^{13})	10.01	93.14	2.61	91.06	3.75	94.65
GAL-MEL (1×10^{14})	13.90	49.82	44.79	58.23	4.01	59.29
ORY-DAN (8×10^{14})	39.44	0.21	11.09	0.15	4.72	0.22
SUS-BOS (7×10^{15})	98.18	5.48	67.78	5.59	8.29	5.00
HOM-MUS (2×10^{16})	179.10	0.96	57.88	0.87	16.58	0.98
HOM-GOR (5×10^{16})	2,098.03	66.29	DNF	DNF	41.52	69.66

5.2.6 Comparison using different Graphics Processing Units

As discussed throughout this manuscript, the seed-and-extend algorithm had not yet been implemented as a GPU algorithm due to the challenges it poses in terms of irregularity. In subsection 5.1.2 "The original GECKO algorithm", the dependencies and boundaries of each stage of the GECKO algorithm were analyzed, showing that in fact, each stage has largely different processing requirements, complexity factors, *etc.* This implies that while a kernel might be bounded by memory bandwidth, another might be bounded by computation, or by host-to-device transfers, *etc.* This can be easily seen by analyzing the memory requirements throughout stages: while the words kernel requires a known amount of memory (see Equation 5.1), the number of seeds is completely unknown and can be multiple times higher than the number of words; in fact quadratically (see seed generation complexity in subsection 5.2.7 "Runtime complexity analysis"). Therefore, making the memory requirements as high as the most demanding step would result in wasted resources during the execution of the other kernels. Moreover, a reasonable assumption would be that increasing the size of the subsequence (see subsection 5.1.3 "Subsequence batching") would reduce runtime by a lot; however this is not the case since in the end it is the number of seeds that dominate the execution (since GPUGECKO has linear complexity) and therefore the additional recomputation of words can be safely ignored (see subsection 5.1.5 "Recomputation in subsequence comparisons"). This comes to show that the elasticity between computation and memory utilization can be further exploited, especially in the case of next generation devices which have much higher memory and processing capabilities, and where the difference between kernels would be much more pronounced.

Therefore, in this section the use of the CUDA Multi Process Service is described in order to fully utilize the potential of the newer NVIDIA cards. The key idea is to split a

large execution into smaller ones (similar to the subsequence batching) in order to enable parallel executions within the same device with each employing the optimal amount of memory. This enables a device to run complementary kernels that are bounded by compute (*e.g.* seed extension) and memory (*e.g.* seed generation) at the same time. The optimal size of global memory per execution was empirically determined to be around 5 to 6 GBs. For the following experiments, the GPU devices described in subsection 5.2.2 "Infrastructure" were used. Notice that these were selected in order to have a representative card from each of the last four generations (at the time of writing), namely Maxwell (980 GTX), Pascal (1080 Ti), Turing (2080 Ti) and Ampere (3090 RTX).

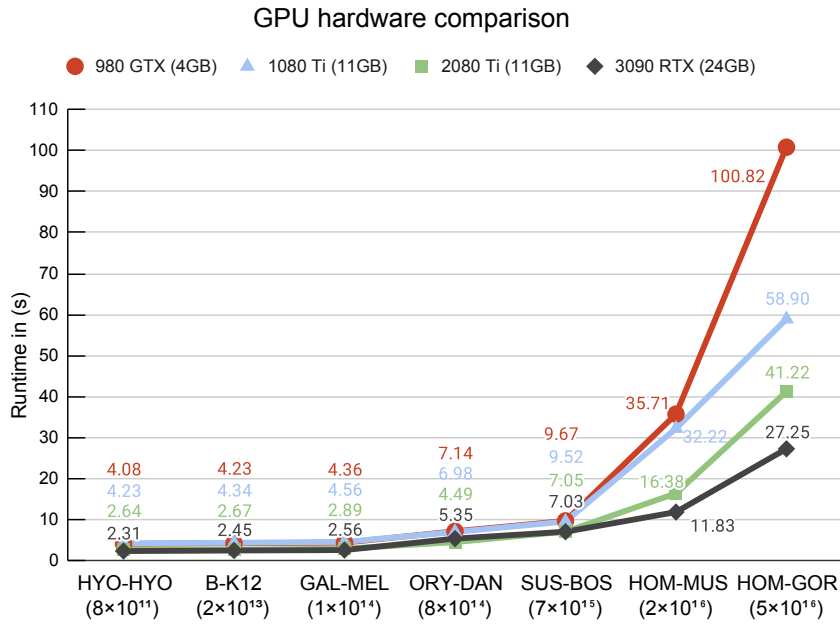


Fig. 5.5 Runtime comparison between a representative GPU card from the last four NVIDIA generations for the sample dataset.

Figure 5.5 and 5.6 show the runtime and speedup, respectively, for each card on the sample dataset. It can be seen that for the smaller datasets (*i.e.* HY0-HY0 to SUS-BOS) the acceleration is limited (below $\approx 2x$); this is a result of the sequences not being large enough in order to take full advantage of the new cards. However, when sequences become larger (*i.e.* HOM-MUS and HOM-GOR) the speedup becomes noticeable, in fact at a rate which correlates to the increase in both cores and bandwidth for the given card. In the case of the HOM-GOR comparison, such rate is an average of 1.55x from one card to the next generation one. This is in agreement with the speedups observed across devices from different generations in [116]. The speedup achieved in the last comparison between the 980 GTX and the 3090 RTX is

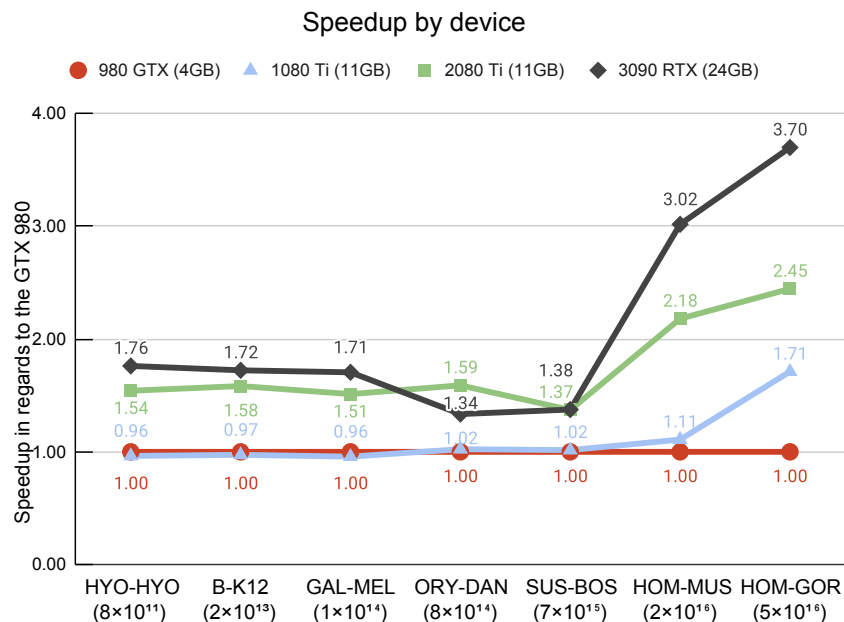


Fig. 5.6 Speedup comparison between a representative GPU card from the last four NVIDIA generations for the sample dataset. The 980 GTX is taken as reference.

3.69x, which is also similar to the difference in bandwidth and number of cores, *i.e.* $\approx 4x$ and $5x$.

5.2.7 Runtime complexity analysis

The runtime complexity of GPUGECKO is analyzed in this section and compared to that of GBLASTN. To do so, an experiment was set up where comparisons for both methods were run in linear increments of the input size. While this is not an exact procedure to generate linear increments of the number of seeds (which are the dominant parameter since they translate into alignments), there is high correlation between input size and output number of alignments. For this particular experiment, the comparison between *Homo sapiens* Chr. X and *Mus musculus* Chr. X was run in increments of 1% of input size while recording the runtime. The results along with the number of seeds generated per input is shown in Figure 5.7.

It can be observed that the runtime complexity of GPUGECKO is tied to the number of seeds. This is expected since all seeds need to be computed, regardless of the fact that all processes involved in GPUGECKO are linear in the size of the input. The first thing to notice is the much higher complexity in terms of runtime of GBLASTN which will be analyzed

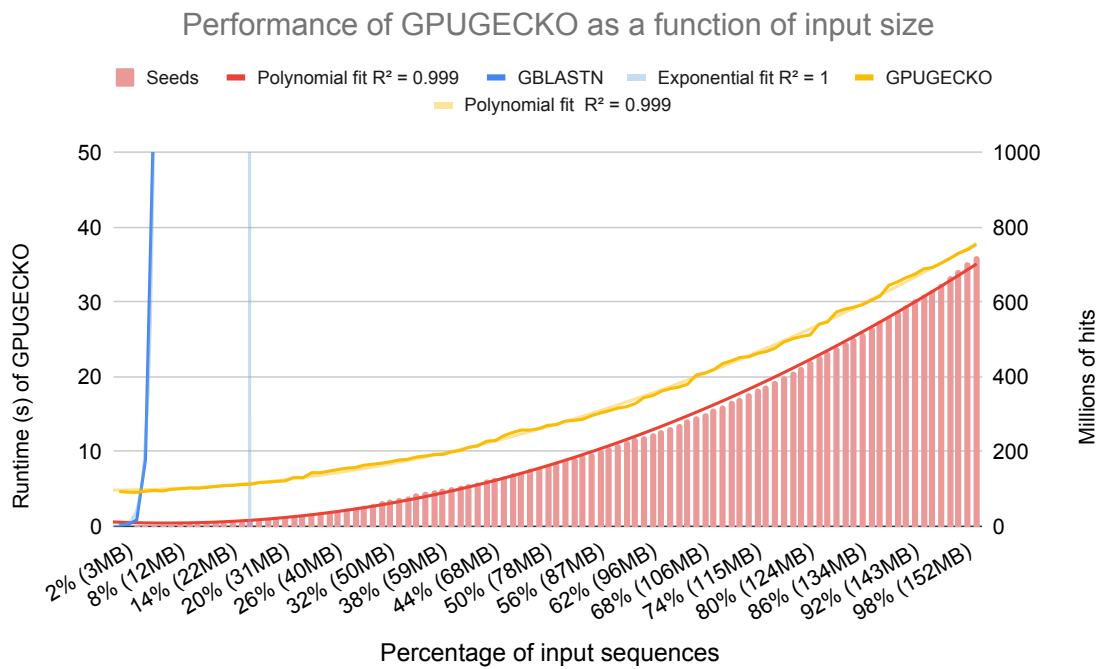


Fig. 5.7 Runtime complexity of GPUGECKO and GBLASTN according to linear increments of the input. The runtime in seconds is shown in the left vertical axis. The right vertical axis features millions of seeds and corresponds to the area (in bars) beneath the red curve of seeds. Polynomial fits are shown for the seeds and GPUGECKO, whereas for GBLASTN an exponential fit was used.

separately. Secondly, the number of seeds follows a nearly perfect second-degree polynomial fit, as described previously in section 2.3.2 "Number of patterns". Thirdly, it is hinted that GPUGECKO has strict linear complexity in the number of seeds. This is further confirmed in Figure 5.8, where a direct fit between the number of seeds and the runtime of GPUGECKO is shown. Moreover, the fit is $y(x) = 0.04x + 5.27$, which shows the initialization period (intercept) and the time cost (in seconds) per processing of a million seeds. In essence, this means that GPUGECKO achieves seed-independence, since the subsequence batching procedure virtually removes memory-related limitations and the small processing overhead per seed guarantees a realistic makespan. In fact, it can be extrapolated that for an execution to last one day (which is still very manageable), a sequence comparison would require almost 2.5×10^{12} seeds, which is well-over three orders of magnitude beyond the largest mammalian chromosomes.

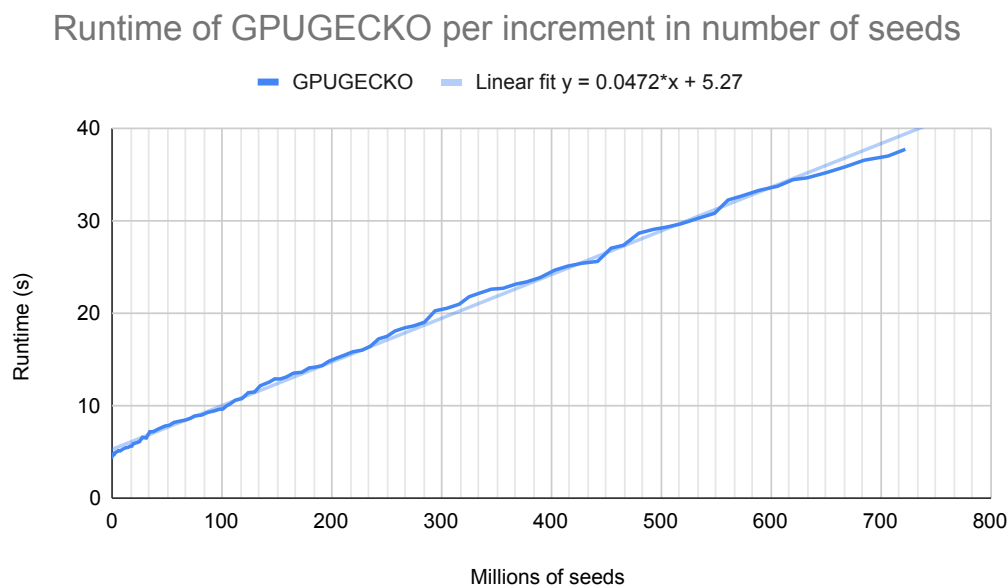


Fig. 5.8 Linear fit between the number of seeds (in millions) and the runtime of GPUGECKO (in seconds).

On the contrary, this is not the case with GBLASTN, which is shown in the left side of Figure 5.7 for the first few datapoints. A closer inspection is shown in Figure 5.9. Its complexity is exponential in the size of the input sequences, as seen both visually and with the exponential fit ($R^2 \approx 1$). The reasons behind this difference in complexity is due to implementation: the largest difference is that in GBLASTN, the reference sequence is scanned in segments (each thread mapping a different region of the reference) and at a first step, the matches are not extended nor saved, and the only information recorded is whether

the reference word has a match or not. This effectively avoids branch divergence, as all threads do nearly the same. However, it comes at the price of performing lots of global atomic operations to record the words. In the case of GPUGECKO, seeds are computed in cooperative lockstep, therefore achieving high efficiency in memory transactions, and synchronization among these threads is performed via shuffle functions, rather than atomic counters. In fact, as described in section 5.1.6 "Exhaustive generation of seeds", atomic counters are only used to sporadically access larger and unowned memory regions when the regular and dedicated ones are exhausted; and still, these calls occur in the count of few thousands (the hard limit is the number of these pools) whereas in GBLASTN it occurs at the level of word matches (although this number is reduced due to the use of local arrays - which also can reduce performance due to register spilling). Further reasons on the complexity difference include:

1. The number of GPU kernels: GPUGECKO runs integrally on the accelerator whereas GBLASTN runs only the most demanding steps.
2. Compute and memory trade-off: while GBLASTN attempts to run a whole comparison at once (which is faster in essence but results in exceeding the limitations of physical memory for medium to large sized sequences), GPUGECKO splits it into batches in an exchange of time versus space.
3. The extension in GBLASTN is performed by giving each thread a different seed. This approach comes with several problems, particularly branch divergence (serialization of operations due to different execution paths) and SM locking (a block will last in the SM as long as one thread has not finished extending resulting in increased latency). In GPUGECKO, this is avoided by running one block of 32 threads per seed in lockstep, thus completely removing both problems (see section 5.1.6 "Extension of seeds").

5.2.8 Large-scale genome comparison study

In a similar fashion to the experiment in subsection 3.2.5 "Massive comparison of complete genomes", a study regarding a massive *all vs. all* comparison is shown next. For demonstration purposes, two comparison maps are shown, namely (1) *Homo sapiens* vs *Mus musculus* and (2) *Homo sapiens* vs *Gorilla gorilla*. Figure 5.10 shows the heatmap for the *Homo sapiens* vs *Mus musculus* comparison. Notice that coverage is computed only for the query, *i.e.* how much of the query can be found in the reference, and it is calculated as the number of shared bases in the alignments (with a minimum length of 128 and 512 for both sequences, respectively) and divided by the length of the query.

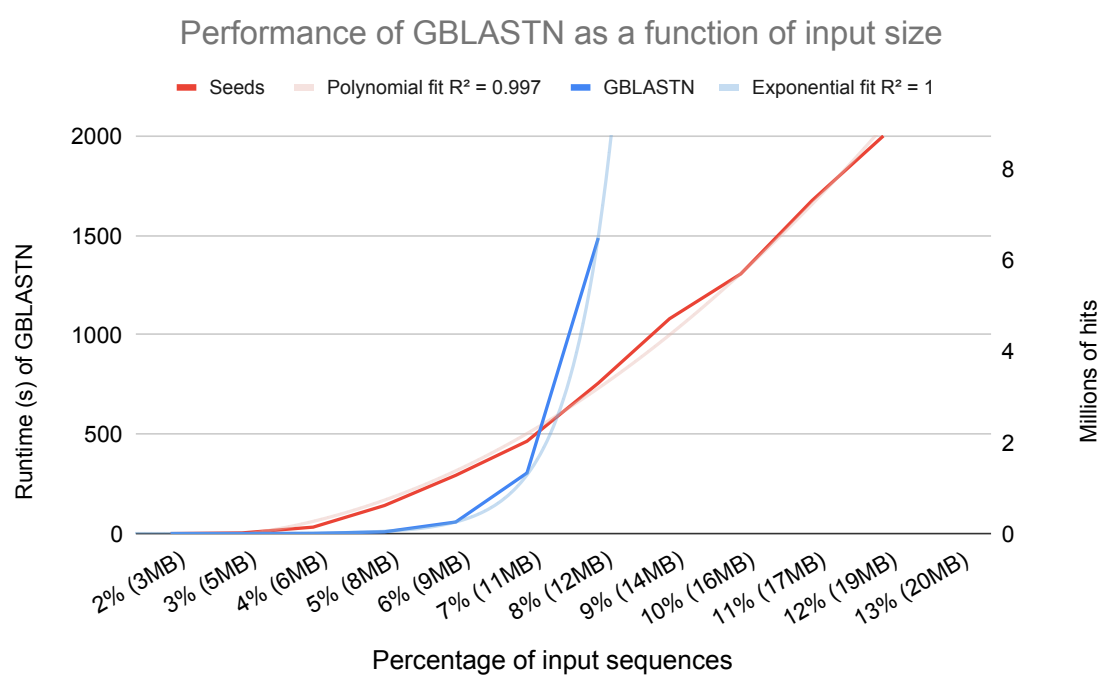


Fig. 5.9 Runtime complexity of GBLASTN according to linear increments of the input. An exponential fit for the runtime of GBLASTN is shown in light blue. The number of seeds (red) at each increment is also shown along with a polynomial fit of degree two.

The exhaustive comparison of *Homo sapiens* vs *Mus musculus*, which included 504 chromosome comparisons ranging in size from 47 megabase pairs to 242 megabase pairs took 3 hours and 39 minutes (an average of 26 seconds per pair of chromosomes comparison), whereas the *Homo sapiens* vs *Gorilla gorilla* took 4 hours and 18 minutes for a total of 600 comparisons. The increase in runtime is due to the much larger number of alignments found between primates, as can be seen in an increase in coverage (see Figure 5.11).

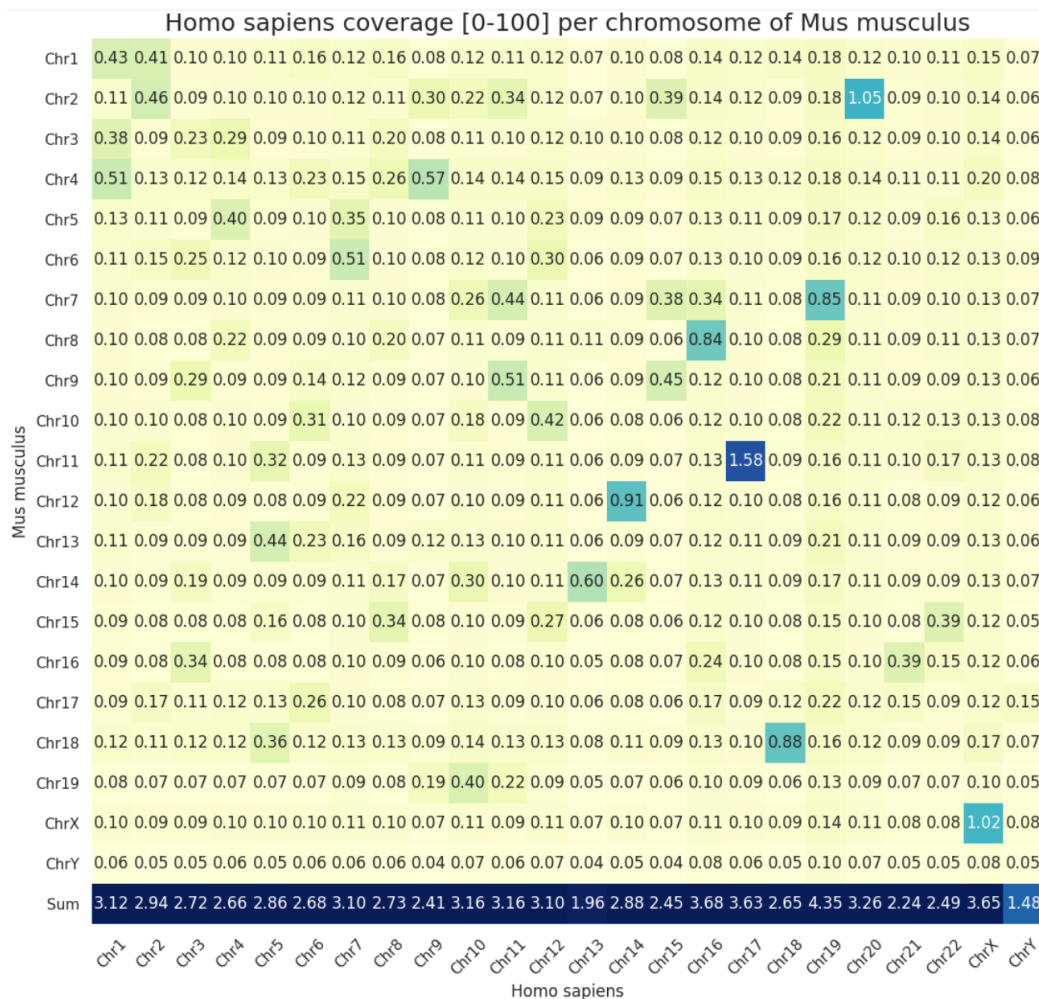


Fig. 5.10 Coverage heatmap of the all-vs-all chromosome comparison between *Homo sapiens* and *Mus musculus*. The coverage is in the range [0, 100]. The last row is the sum of the coverage per each human chromosome which is **not** in the range [0, 100] but rather [0, $n * 100$] where n is the number of chromosomes of *Mus musculus*.

As can be seen, less evolutionary rearrangements have occurred between *Homo sapiens* vs *Gorilla gorilla* compared to *Homo sapiens* vs *Mus musculus*, as expected given the phylogenetic studies in literature.

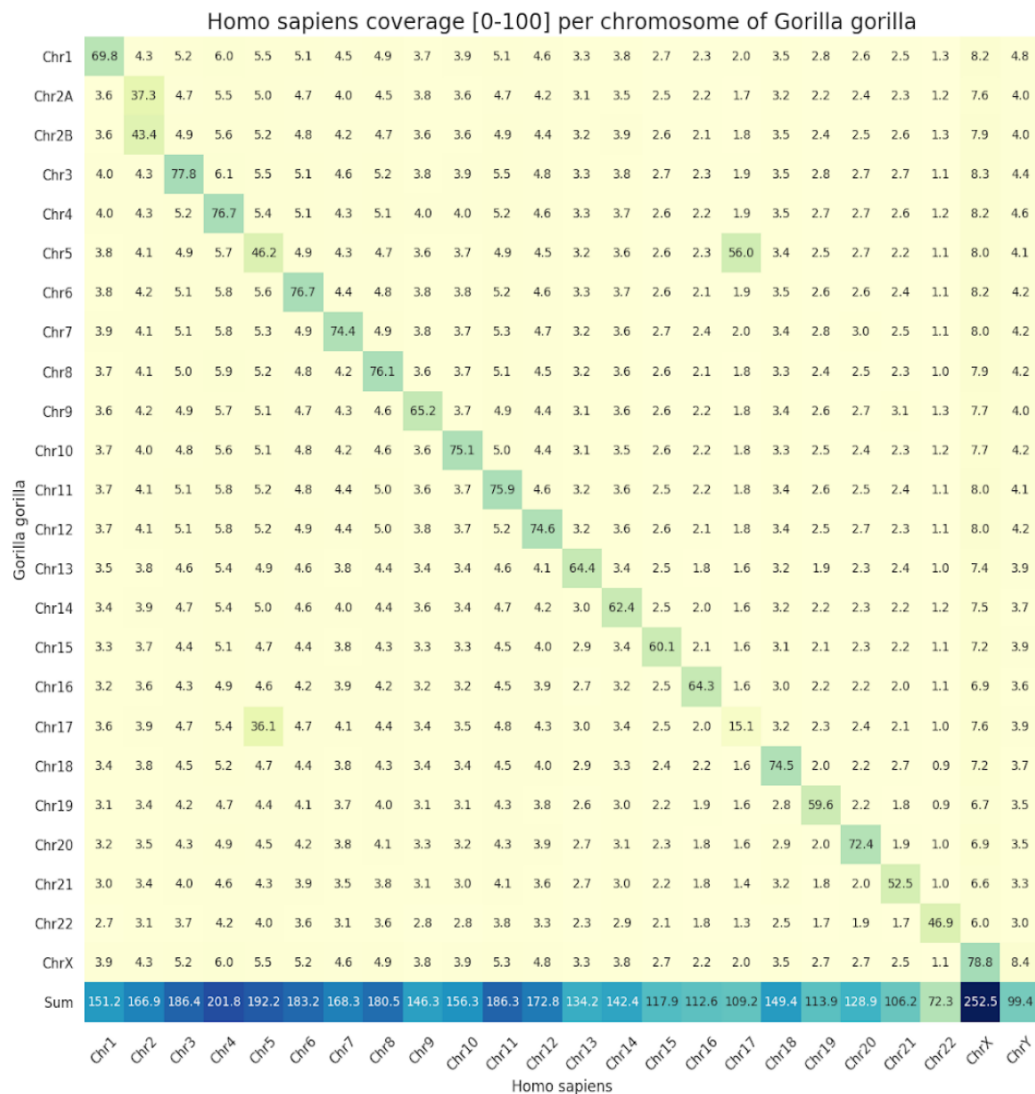


Fig. 5.11 Coverage heatmap of the all-vs-all chromosome comparison between *Homo sapiens* and *Gorilla gorilla*. The coverage is in the range $[0, 100]$. The last row is the sum of the coverage per each human chromosome which is **not** in the range $[0, 100]$ but rather $[0, n * 100]$ where n is the number of chromosomes of *Gorilla gorilla*.

Chapter 6

Scheduling of genome comparisons using Machine Learning

6.1 Methods

6.1.1 Introduction

The previous chapters of this manuscript have covered the perspective of accelerating a particular application with the aim of speeding up the computation of both coarse and fine-grained sequence alignment. This is without a doubt one of the most common approaches of applying advanced computing to a particular area. However, the discussed proposals only applied to improving the performance of a single task, and, needless to say, High Performance Computing is also related to all of the elements that conform a concurrent system, such as schedulers, dispatchers, resource controllers, *etc.* Among these parts, the scheduler is one of the most important processes, since it is in charge of deciding which tasks should be executed with higher priority, and which ones should be put to wait. Such a decision can become quite difficult, particularly in the case of cloud-computing clusters where many users can send jobs simultaneously and therefore resources need to be scaled up or down on demand, which potentially leads to increasing runtime costs, waste of resources, performance penalty, *etc.*

Traditional scheduling approaches include, to cite some, the well-known First-In First-Out (FIFO) [117] or Round-robin [118] techniques. However, nowadays most schedulers are based on a compound of several metrics as opposed to one-size-fits-all policy ones. These can include traditional ones such as the walltime of the job or memory requirements, which are typically submitted before runtime as part of the configuration file, or other ones more system-wide related such as user penalty and/or execution priority based on their previous job submissions [119]. Another common technique to improve performance is known as

backfill [120], which is aimed at introducing lower-priority jobs among higher-priority jobs without affecting their execution.

Nevertheless, these techniques rely on the resource requirements estimated by users, which has been shown to be inaccurate [121]. Therefore, being able to accurately estimate the resource consumption of a process is key to increasing the optimality of job schedulers, and therefore the throughput and latency of a concurrent system. A particularly well-suited example is in fact the field of sequence comparison due to the unknown number of similar elements between sequences, which, as shown in Table 5.3 "Number of seeds per pairwise comparison" and section 5.1.6 "Generation of seeds", can be quadratic in the worst case and therefore govern runtime.

The problem of estimating the evolution of DNA sequences has been researched thoroughly, for instance in the form of analytical studies where the parameters of the distributions are calculated and bounded [122], or by computationally estimating the probabilities of nucleotide transitions, such as with Markov chains [123]. However, these methods are not oriented to accurately estimate the number of word-matches, but rather to understand the evolution of DNA sequences along the passing of time.

In the following sections, a holistic approach based on using the frequencies of small DNA words (which can be computed in strict $O(n)$ time) to serve as input for a Machine Learning model which then predicts the number of matches of larger words, is presented. This can be in turn be used to estimate the amount of computational resources required to allocate for other sequence comparisons, thus improving the throughput of supercomputers dedicated to Comparative Genomics experiments such as the NCBI [124] or the EBI [125] portals.

It must be mentioned that at the time of writing, this chapter is still an on-going work (and therefore further extensions are expected) which has been published only at a national conference, namely:

- Pérez-Wohlfeil, Esteban and Oswaldo Trelles. "Towards the efficient scheduling of sequence comparisons using Machine Learning", Jornadas Sarteco 2021, VI Congreso Nacional de Informática (CEDI), Málaga, Spain, Sep 2021.

6.1.2 Formulation

The mathematical formulation for DNA sequences and the number of similar word matches was described in subsection 2.3.1 "Definitions". Equation 2.3 introduced how to calculate the numerical value associated with a fixed-size DNA substring. The following sections extend on this definition. Once all words w_i have been computed for two sequences q and r (query

and reference), and given the counts $c_{i,q}$ and $c_{i,r}$ for each word w_i , the number of similar fixed-words or hits h is equal to:

$$h = \sum_{i=0}^{|V|^k} (c_{i,q} * c_{i,r}) \quad (6.1)$$

where $c_{i,q}$ is the number of words of size k which have hash value equal to i in the query sequence (the same applies to $w_{i,r}^k$ but in the reference sequence). Therefore, predicting the number of hits h of size k is simply a summation over a multiplication. However, due to the exponential term in the summatory $|V|^k$, it is not possible to index the number of words of large k in strictly $O(n)$ time, since the memory requirements become too high and hence resorting to complex indexing data structures is required (see section 2.3.3 "Data structures"). In fact, this is one of the mostly-addressed problems in bioinformatics, known as k -mer counting problem [126, 127].

For k values ≥ 16 (approximately), which are typical in large sequence comparison experiments, the number of hits is not available unless a considerable amount of computation is performed. Therefore the question to be answered is whether it is possible to estimate h based on the frequencies of lower k -mer words which can be computed in strictly $O(n)$ time as a result of reduced memory requirements. This is equivalent to finding a function f that maps the frequencies of lower-order words to higher-order words (*i.e.* from smaller to larger k sizes) and then minimizing the difference between the number of hits for the high-order k and the lower-order one.

In this section, certain machine learning models will be studied in terms of their ability to generalize such mapping function f using the lower order of $k = 3$ and the higher order of $k = 32$. This effectively means that the input to the machine learning model will be the k -mer frequencies of size three (trinucleotides, *e.g.* words ATT or CGA) of two sequences being compared and the targets will be the real number of hits of size $k = 32$. Hence the model needs to learn both a generalization of the frequencies as well as their summation and product, resulting in a multidimensional regression problem. On one hand, the value of $k = 3$ for the input frequencies was chosen because it offers an adequate trade-off between computation requirements, probability of not finding null-mers [128, 37] and finally because amino acids are coded using triplets of nucleotides, which could also enable to combine nucleotide and codon information simultaneously in the training stage of the models. On the other hand, the value of $k = 32$ was chosen since it is a common k -mer size employed in large-scale chromosome comparisons and due to it being sufficiently far away from the

lower-order value of $k = 3$. Nevertheless, other k values (both smaller and higher) could have been used for the higher order.

For the context of scheduling systems and the evaluation of results, the problem is simplified by transforming it into a classification problem between three classes, namely whether the sequence comparison produces *few*, *some* or *many* hits. Further details are given in the following sections.

6.1.3 Overview of the methodology and datasets

The proposed study begins with the data retrieval (downloading DNA sequences) and generation (running sequence comparisons), continues with the model training and evaluation (predicting on new sequences) and finishes with the throughput simulation. Each step is described in the following sections.

Predicting the number of hits in small sequence comparisons does not pose as much interest as in the case of large sequences, since current methods are perfectly able to deal with sequences of few megabase pairs. Therefore, this study is limited to chromosomes of complex organisms of at least 1 MBp of size (and up to 200-300 MBp mark). Given the recent increase in whole genome sequence availability, over 2,675 sequences from 110 different genomes were fetched featuring mammals, fish, birds, amphibians and reptiles. This amounted to over 190 GB in sequence data. Only fully assembled sequences at chromosome level were used. Table 6.1 shows the different types of species employed in the training dataset, and Table 6.2 shows the species used for the validation dataset.

The number of possible comparisons is $2,675 * 2,675 = 7,155,625$. If each comparison took 10 minutes (typically it requires more on traditional CPU algorithms), the required experiments would consume 1,192,604 CPU hours. Therefore, in order to leverage computing time, GPUGECKO was employed (see chapter 5 "GPU-acceleration for the irregular and exhaustive sequence comparison") along with a sampling procedure. The sampling procedure consisted on randomly selecting which sequences would be compared (as opposed to comparing all against all). While the full computation is still preferable, a sample that resembles the population can be obtained with enough computing time.

In total, 210,000 chromosome-sized sequence comparisons were performed, which were split into 180,000 for the training set and 30,000 for the validation set. Note that because of the small amount of species available (110, as mentioned earlier), a further split between training, validation and testing datasets was not possible, since this would have caused huge imbalance between the proportion of taxonomic ranks of the species, *i.e.* having many more species of one family than other.

Table 6.1 Genome species included in the training dataset by genomic class.

Genomic class	Species included
Mammalia	Homo sapiens, Microcebus murinus, Callithrix jacchus, Chlorocebus sabaeus, Macaca mulatta, Pan paniscus, Mus musculus, Bos taurus, Equus caballus, Canis familiaris, Sus scrofa, Oryctolagus cuniculus, Rattus norvegicus, Ovis aries, Mus spretus, Camelus dromedarius, Zalophus californianus, Lynx canadensis, Felis catus, Macaca fascicularis, Canis lupus basenji, Domestic yak, Sciurus vulgaris, Theropithecus gelada, Ailuropoda melanoleuca, Nomascus leucogenys, Capra hircus, Gorilla gorilla, Bos taurus and indicus hybrid, Panthera leo, Pongo abelii, Meleagris gallopavus, Suricata suricatta, Mus musculus castaneus, Peromyscus maniculatus bairdii, Papio anubis, Monodelphis domestica, Ornithorhynchus anatinus, Microtus ochrogaster, Mus caroli, Mus pahari, Phryseter catodon, Cervus hanglu yarkandensis, Phocoena sinus.
Aves	Gallus gallus, Anas platyrhynchos, Aquila chrysaetos chrysaetos, Parus major, Strigops habroptila, Cairina moschata domestica, Malurus cyaneus samueli, Camarhynchus parvulus, Taeniopygia guttata.
Actinopterygii (fish)	Danio rerio, Clupea harengus, Asian arowana, Takifugu rubripes, Carassius auratus, Larimichthys crocea, Astyanax mexicanus, Amphiprion percula, Sphaeramia orbicularis, Lepisosteus oculatus, Tetraodon nigroviridis.
Reptilia	Chrysemys picta bellii, Anolis carolinensis, Podarcis muralis, Gopherus evgoodei, Naja naja.
Amphibia	Leptobranchium leishanense, Xenopus tropicalis.

Table 6.2 Genome species included in the validation dataset by genomic class.

Genomic class	Species included
Mammalia	Giraffa camelopardalis rothschildi, Tursiops truncatus, Cervus elaphus, Equus asinus, Vicugna pacos, Bos indicus, Bubalus bubalis, Cricetulus griseus, Canis lupus (wolf), Mustela erminea, Ptilocolobus tephrosceles, Rhinopithecus roxellana.
Aves	Cairina moschata domestica, Coturnix japonica, Cariama cristata, Columba livia, Corvus cornix cornix, Cygnus olor, Dromaius novahollandiae, Lonchura striata, Numida meleagris.
Actinopterygii (fish)	Salmo salar, Acipenser ruthenus, Cyprinus carpio, Epinephelus fuscoguttatus, Melanotaenia boesemani, Oncorhynchus mykiss, Poecilia reticulata, Pygocentrus nattereri.
Reptilia	Trachemys scripta elegans.

6.1.4 Comparisons with GPUGECKO

Generating the sequence comparison is the most time-consuming stage. This is due to two things, (1) as many comparisons as possible should be performed and (2) the comparisons need to be exhaustive, in the sense that the sequence comparison should produce the real number of word-matches (hits h in Equation 6.1) and not a pruned, heuristically filtered subset (as in other algorithms, see section 5.1.6 "A note regarding seed-generation in state-of-the-art methods" algorithms), or otherwise the learning model will be subjected to this bias.

For such reasons, GPUGECKO was used to reduce runtime and memory requirements while exhaustively producing all possible alignments. GPUGECKO was used to output the total number of hits (instead of the alignments) of size $k = 32$ and this value was used as the number of word-matches in every comparison. In addition, each sequence comparison was augmented by providing the reverse complement of the reference sequence, thus generating double the amount of data. This is possible without requiring to recompute the sequence comparison since the number of hits remains the same. The frequencies of words of size $k = 3$ were then calculated and saved for each comparison. Figure 6.1 shows a sample of trinucleotide frequencies for several species ranging multiple genomic classes.

It can be observed from Figure 6.1 that the frequency of trinucleotides (*i.e.*, the input to the models) is mostly conserved even across remotely related species, implying that a simple prediction based on huge differences between relative frequencies is not enough to accurately predict the number of larger-order k -mers.

6.1.5 Data preprocessing

The input to the models is a vector containing the m frequencies for the trinucleotide words of each sequence, *i.e.* 64 values per sequence at two sequences per comparison, thus equalling $2 * 64 = 128$ frequency values per each training sample. The targets are a vector containing the class to which the number of hits were assigned. Three classes were devised: *few*, *some* and *many*, corresponding to the thresholds $h_s \leq 10^6$, $10^6 < h_s \leq 10^8$ and $10^8 < h_s$. This partitioning arises naturally from the targets distribution, keeps the classes relatively balanced and finally also serves its purpose of assigning predefined sets of computational resources to each class.

Out of all sequences, two main datasets were created for training and validation purposes. The training dataset comprises 71 genomes whereas the validation dataset 31 genomes. The validation dataset is further split in additional datasets as shown in Table 6.3. These are used to illustrate the generalization capabilities of the model depending on whether the model is

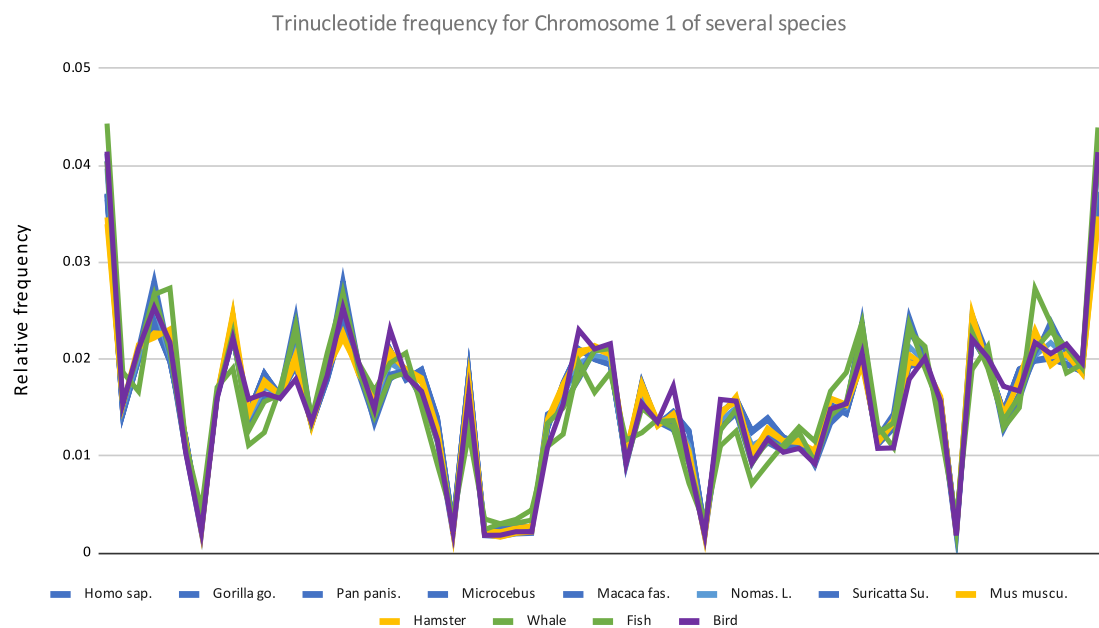


Fig. 6.1 The relative frequency of trinucleotides from multiple species across different genomic classes is shown. The x-axis contains the 64 possible trinucleotides (3-mers), starting from the left with AAA and ending on the right with TTT. The y-axis represents the relative frequency of each trinucleotide such that the sum of all of them is equal to one. The labels *Fish* and *Bird* correspond to the genomes *Sphaemaria orbicularis* and *Malurus cyaneus*, respectively.

Table 6.3 Validation datasets sorted by their novelty level. This level refers to which part of the dataset the model has seen previously (either the sequences or the result of the comparison, or none).

Validation dataset	Query sequences	Reference sequences	Targets
Unknown vs. unknown	Unseen	Unseen	Unseen
Unknown vs. known	Unseen	Seen	Unseen
Known vs. known	Seen	Seen	Unseen
Fully seen	Seen	Seen	Seen

familiar with the sequences or the actual comparisons. For instance, in dataset Unknown vs. unknown, the model has never seen the sequences in the query nor the reference, and therefore the comparison target (the resulting number of hits) is new to the model. On the contrary, the Unknown vs. known dataset features unknown queries but known references, however the comparison target is still new since it depends on both sequences. Lastly, the Known vs. known dataset includes sequences which have been used during training but whose comparison has not been seen by the model. These datasets can then be used to assess the level of generalization of the network depending on how novel the sequences are. For example, in a scenario where a model is retrained as new comparisons arrive, it is likely that some sequences will have already been seen by the model; but this does not imply that both sequences have been seen by the model or that their comparison has been performed previously already.

6.1.6 Normalization

Prior to feeding the data to the models, the raw frequency vectors (k -mer frequencies) were normalized using the min-max algorithm. The targets were one-hot encoded into three classes following the thresholds discussed previously. Correlations in the input data are expected due to the simple fact that the probability of observing a word $w = x_1, x_2, x_3$ depends on the probability of observing the previous nucleotides x_1, x_2 , *i.e.* $P(w) = P(x_3, x_2, x_1) = P(x_3|x_2, x_1) * P(x_2, x_1)$. Furthermore, half of the frequency variables can be removed without sacrificing any accuracy, since there is a nearly perfect correlation between complementary nucleotides, *e.g.* between AAA and TTT.

It is noteworthy to mention that the probability of observing $(x_i|x_{i-1}, x_{i-2}, \dots, x_0)$ cannot be estimated from the previous probabilities $x_{i-1}, x_{i-2}, \dots, x_0$, or otherwise knowing the initial probabilities $P(a, c, g, t)$ would suffice to accurately calculate the probability of composed strings. This is shown in Table 6.4, where the initial probabilities were measured for *My-*

Table 6.4 Example of non-uniform probability distribution of the frequencies of words of size $k = 2$ of the sequence *Mycoplasma genitalium* G37. The first and third column show the dinucleotide for which frequencies are estimated, the second and fourth columns corresponds to the actual probability as calculated experimentally and the third and sixth column show the estimated probabilities by multiplying the corresponding initial probabilities.

$x_i x_j$ (1-8)	$P(x_i x_j)$	$P(x_i) * P(x_j)$	$x_i x_j$ (8-16)	$P(x_i x_j)$	$P(x_i) * P(x_j)$
AA	0.146	0.120	GA	0.050	0.055
AC	0.052	0.055	GC	0.030	0.025
AG	0.058	0.055	GG	0.028	0.025
AT	0.089	0.117	GT	0.051	0.054
CA	0.063	0.055	TA	0.087	0.117
CC	0.029	0.025	TC	0.047	0.053
CG	0.010	0.025	TG	0.063	0.054
CT	0.056	0.053	TT	0.140	0.114

coplasma genitalium G37 to be $a = 0.346c = 0.158, g = 0.159, t = 0.337$ and the compared to the multiplicative ones of size two. It can be seen that even at single step increase in the estimated frequencies (from single nucleotides to dinucleotides), the error can be as large as two and half times the probabilitiy ($P(c|g = 0.010)$ whereas $P(c) * P(g) = 0.025$).

6.1.7 Evaluated models

Three different methods were trained and evaluated. These are (1) random forest, (2) xgboost and (3) a residual neural network. Support Vector Machines were not considered due to their computational performance when training on large datasets [129]. Although the first two methods, namely random forest and xgboost do not require normalization of the input data, they were trained on the same normalized data as the neural network since this does not affect the outcome predictions. The best parameters for both were found using a grid search approach. Table 6.5 shows the parameters of the grid search employed for each of the two methods.

The design of the neural network was based on multi-layered residual networks [130] and particularly on an architecture specifically designed for non-linear regression [131], since the depicted scenario is largely a regression problem coupled with a classification layer at the end. The number of layers and the width were determined experimentally through a series of grid searches, resulting in a neural network with 6 residual blocks, each containing three dense and batch normalization layers [132] of a fixed 64 neurons each

Table 6.5 Hyperparameters set in the grid search corresponding to the learning models Xgboost and Random forest.

Xgboost param.	Xgboost values	Random F. param.	Random F. values
'objective'	'multi:softmax'	'bootstrap'	True
'num_classes'	3	'criterion'	['gini', 'entropy']
'gamma'	[0.0,0.3,0.6,0.99]	'max_features'	'auto'
'learning_rate'	$10^{-2} * [0.1,1,3,5]$	'max_depth'	[5,15,30]
'max_depth'	[3,5,7,10]	'min_samples_leaf'	[1,3,5]
'min_child_weight'	[1,3,5]	'min_samples_split'	[1,10,25,50]
'subsample'	[0.3,0.5,0.7]	'n_estimators'	$10^2 * [1,5,10, 50,100]$
'colsample_bytree'	[0.3,0.5,0.7]		
'n_estimators'	[100,1000]		

(corresponding to the number of features divided by two due to existing correlations, twice the number of trinucleotides). The design of the network is shown in Figure 6.2, where only a single residual block is shown along with its inner dense and batch normalization layers and the outer skip layer. The activation functions used were rectified linear units (ReLU) [133] and the categorical crossentropy loss was optimized using the ADAM optimizer [134]. Additionally, early-stopping procedures were set up to limit training runtimes.

6.1.8 Scheduling simulation

The proposed models will be evaluated using two simulations of supercomputer traces in order to quantify the impact on throughput of the designed prediction system.

The devised simulation makes use of a FIFO queue to assign priorities to each submitted job, and is comprised of a single node accounting for all resources (therefore no distributed simulation is performed). The node is modelled including 512 cores and 4,096 GB of shared memory. Each submitted job consumes only one core, but more RAM and time if the job required more resources (higher allocating class). Jobs were designed to consume the following amount of resources:

- *Few* class (c_0): one core, 100 seconds and 8 GB of RAM.
- *Some* class (c_1): one core, 1,000 seconds and 64 GB of RAM.
- *Many* class (c_2): one core, 10,000 seconds and 512 GB of RAM.

Note that (1) multicore execution could have also been included in the simulation, however for the sake of simplicity it was kept to single execution in order to only have

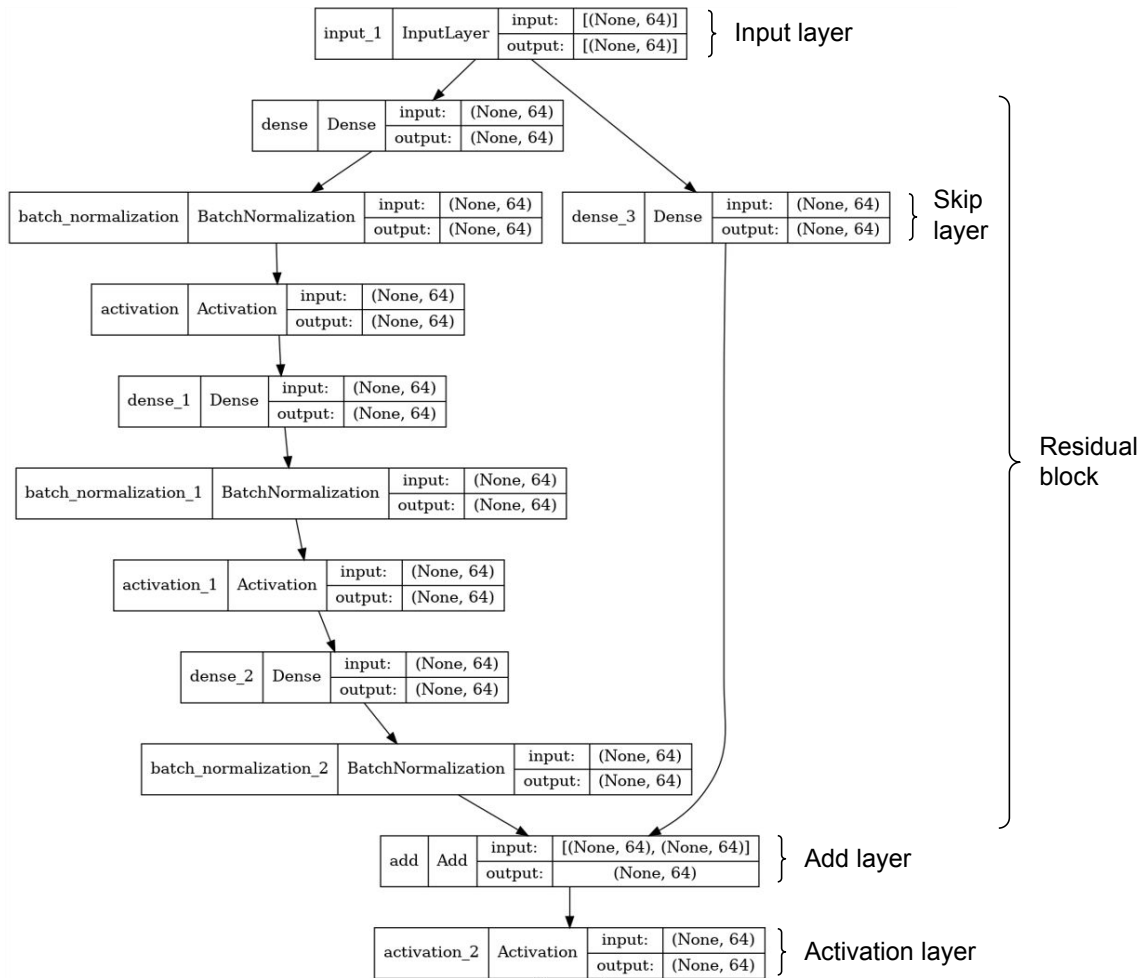


Fig. 6.2 Residual neural network architecture featuring a single residual block, the inner dense and batch normalization layers along with the skip layer for the residual connection, which are gathered in an addition and activation layer at the end. Note that the employed network uses multiple residual blocks. The numbers 64 correspond to the input and output dimension of the layers, which is maintained during the remaining blocks.

runtime affect the elapsed time directly, and (2) due to the nature of the number of seeds, the runtime could have been adjusted quadratically per each class. Nevertheless, because there is also an increase in space, it was decided to increment it by 10 in each class.

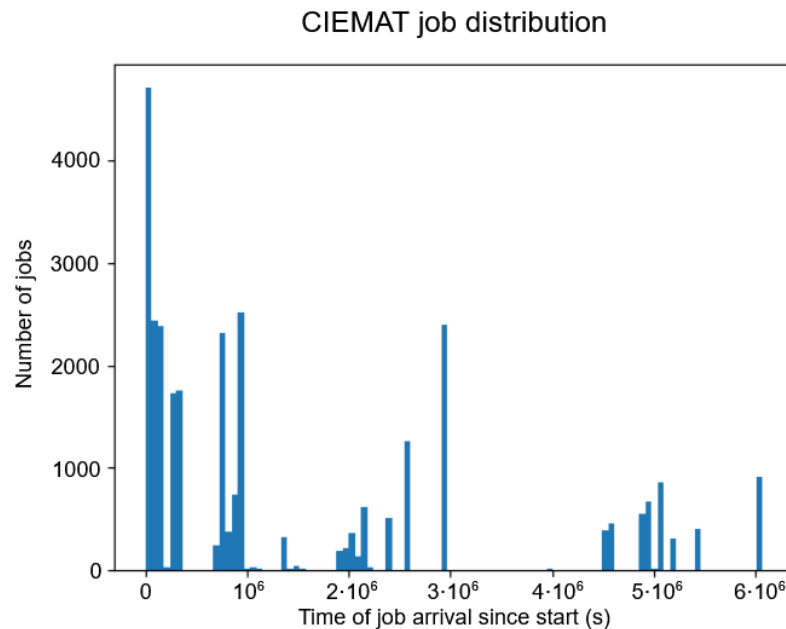


Fig. 6.3 Job submission distribution of the CIEMAT supercomputer since the beginning of the trace. The x-axis shows the number of seconds elapsed since the start of the trace. The y-axis shows the number of jobs submitted at the current second.

The simulation works as follows: a job executed with less resources than required will crash halfway through the execution and then require a relaunch with the immediately superior resource class. The new time of submission for the relaunched job followed a normal distribution $N(10, 4)$ (in hours) in addition to the timestamp of the crash.

Lastly, two workload distributions are considered, one which spaces jobs uniformly throughout the time frame and one following the submission times of the first 30,000 jobs submitted to the CIEMAT supercomputer [135]. These jobs correspond to the size of the test dataset Unknown vs Unknown (see Table 6.3), whose values are used in the evaluation in subsection 6.2.5 "Supercomputer scheduling simulation". The distribution of submitted jobs for the CIEMAT supercomputer is shown in Figure 6.3, which exhibits the standard practice of batching array jobs in specific points in time. This trace was selected for such a behaviour, since it is in this type of scenarios when accurate scheduling can have a greater impact in throughput.

6.2 Results

6.2.1 Introduction

In the following sections, five experiments are performed to compare the different models, validate the learning approaches and lastly study the interpretability of the predictors in the context of computational biology. These include:

1. Training and validation comparison by model.
2. Discussion on precision, recall and maximization of predictions.
3. Supercomputer scheduling simulation using the different models.
4. Explaining predictions via contribution analysis.
5. Predictions as a regression problem.

6.2.2 Infrastructure

All sequence comparisons were performed using the GPU cluster of a local server at the University of Malaga (namely the same server as in subsection 5.2.2 "Infrastructure") with 2x Intel Xeon E5-2698 v3 processor at 2.3 Ghz, 16 cores each. The system included 256 GB of RAM and 4x NVIDIA GeForce GTX 980, 4 GB of global memory.

The training and evaluation of models was run in another local server with 2x 8-core Intel Broadwell-EP E5-2620V4 processor at 2.1 Ghz, 64 GB of RAM and 2 TB of local storage. This is the same server as in Section 3.2.1.

6.2.3 Training and validation comparison of the models

The accuracy of the three models (random forest, Xgboost and neural network) for the mixed dataset (see Table 6.1) is shown in Table 6.6. This table depicts the generalization level of the methods in decreasing importance from top to bottom.

From Table 6.6 it can be observed that the neural network is able to generalize significantly better, achieving 81% accuracy on the completely unknown validation dataset (where neither frequencies or seed numbers have been previously seen in the training stage) compared to 66% and 69% of the random forest and boosting method. In fact, both the boosting and random forest methods are unable to generalize enough regardless of an increase in the complexity of the models, as seen in Figure 6.4 and 6.5.

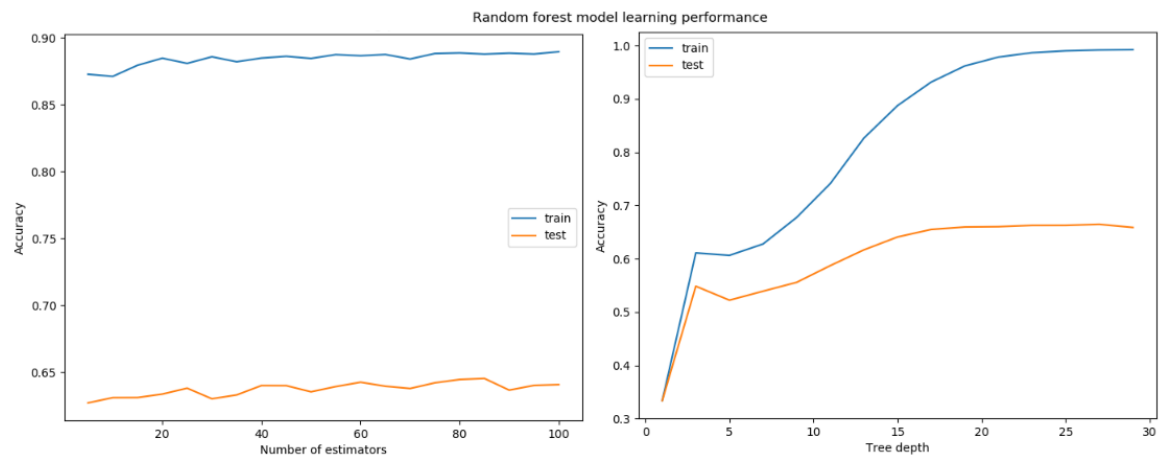


Fig. 6.4 Learning curve for the training and test datasets for the Random forest model by number of estimators (left) and tree depth (right).

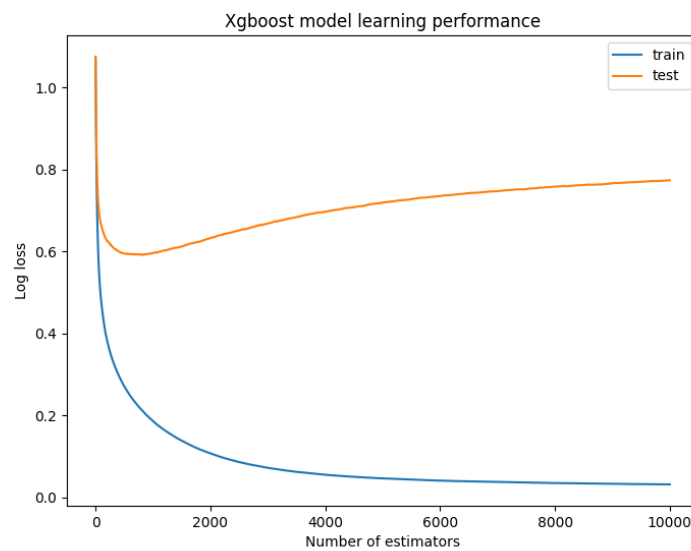


Fig. 6.5 Log loss curve for the training and test datasets for the Gradient boosting model by number of estimators.

Table 6.6 Model comparison. The first column represents the validation dataset. The rest of the columns are the accuracy of the models per each dataset, namely (1) Random forest, (2) Gradient boosting (XGBoost) and (3) Residual neural network.

Validation dataset	Random Forest	Gradient Boosting	Residual Net
Unknown <i>vs</i> unknown	0.66	0.69	0.81
Unknown <i>vs</i> known	0.77	0.79	0.82
Known <i>vs</i> known	0.80	0.84	0.87
Fully seen	0.99	0.94	0.88

A significant difference in accuracy between the Unknown *vs* unknown and the fully-seen dataset is expected (especially in the random forest and gradient boosting methods, which achieve less generalization) due to the variance between the training and evaluation datasets. This is potentially a consequence of not having the exact same proportion of species by genomic class in the validation dataset when compared to the training dataset (see Table 6.2, which is indirectly caused by (1) the limited number of fully-assembled genomic sequences available at the time and (2) the computational resources and time available to perform the study. Although techniques such as k -fold validation [136] could be used to reduce variance, the proposed neural network model shows already promising behaviour, particularly given that more and more high-quality genomic sequences will be made available in the near future.

In fact, this is further confirmed by splitting the validation dataset by genomic species, *i.e.*, setting aside mammalian genomes from aves, fish and reptiles. Despite all efforts in balancing the training and validation datasets (including class weights), the number of mammalian genomes is significantly higher than that of aves or fish. Therefore, the neural network was also trained on each of these datasets individually to find out whether predictions could be improved if the variance between datasets was lowered.

Figure 6.6, 6.7, and 6.8 show the learning curves for both training and testing in terms of accuracy for the three largest dataset splits, namely mammalian, bird and fish genomes.

In the case of the mammalian dataset, the variance between training and test is minimal. Along with sufficient data, the network is able to achieve 94% accuracy on the training dataset and 92.5% on the validation dataset (comprised only of mammalian species, see Table 6.2). Notice that it takes the network around 400 epochs to reach the maximum prediction accuracy.

In the case of the fish and aves datasets (see Figure 6.8 and 6.7), a higher variance is observed between both datasets, especially in the fish dataset, where little is learnt by the

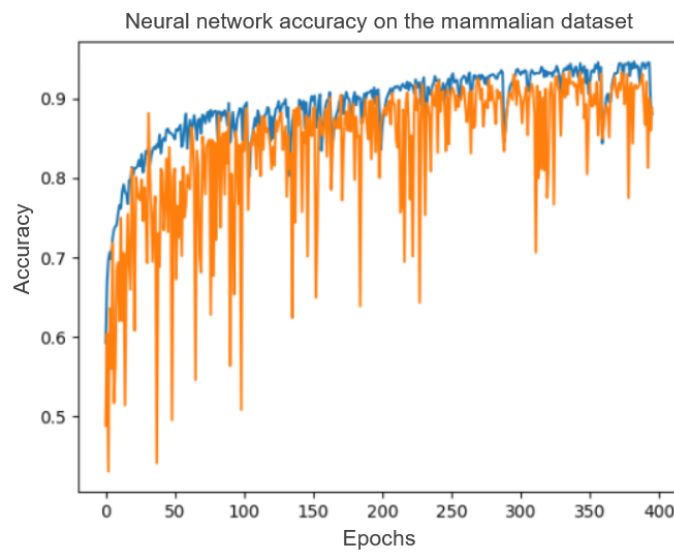


Fig. 6.6 Learning curve of the neural network for the mammalian dataset as function of the number of training epochs. The training accuracy is shown in light blue, slightly above the validation shown in orange.

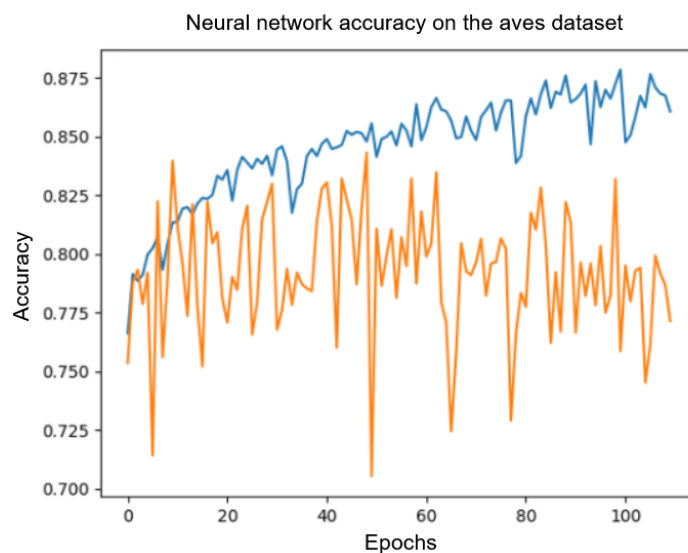


Fig. 6.7 Learning curve of the neural network for the aves dataset as function of the number of training epochs. The training accuracy is shown in light blue, slightly above the validation shown in orange.

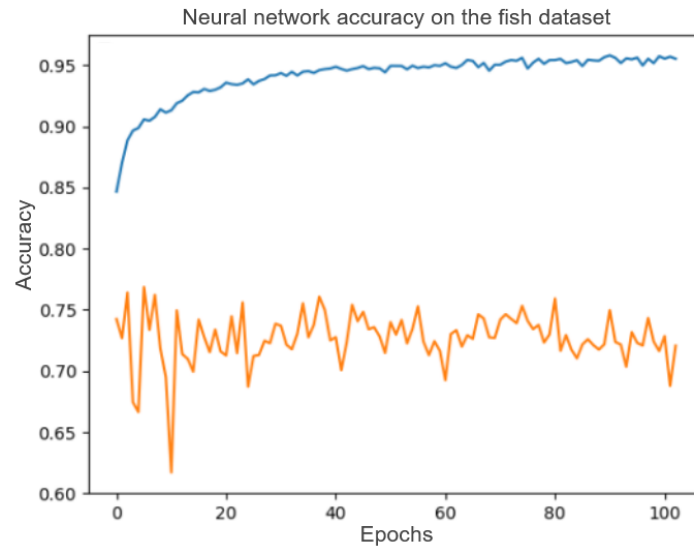


Fig. 6.8 Learning curve of the neural network for the fish dataset as function of the number of training epochs. The training accuracy is shown in light blue, slightly above the validation shown in orange.

model after the first epoch. Despite the high variance and clear signs of overfit in both datasets, the network achieves a 70.5% and 76% validation accuracy, respectively.

In theory, given more and higher quality genomic sequences of the fish and aves dataset, the network should be able to generalize just as well as with the mammalian dataset.

6.2.4 Precision and recall

The overall accuracy of the neural network is approximately 81%, measured as the number of correctly classified instances divided by the total number of instances. While the accuracy metric gives a good understanding of the performance of the model, it must also be discussed the fact that wrong classifications can have more or less impact depending on which class was incorrectly identified. Just as in traditional medicine, sometimes false positives are preferred over false negatives, depending on the effect of the treatment on patients. Therefore, the issue of which metric to maximize (besides accuracy) *i.e.* precision or recall is discussed.

On one hand, misclassifying a class c_2 instance (*i.e.*, a more demanding comparison) is the worst scenario, since it would probably end in the comparison process crashing due to lack of resources, thus needing a restart and consuming more time to recompute all processing which was lost.

On the other hand, misclassifying a class c_0 (*i.e.*, the lowest resource demanding comparison) instance would result in wasting computational resources, however the outcome

would not be as badly as the previous case since the comparison would finish without error, therefore requiring no need for recomputation. This means that for class c_2 a high recall is desired (correctly detected class c_2 instances $\div$ total class c_2 instances), whereas for class c_0 high precision is sought (correctly detected class c_0 instances $\div$ total predictions in class c_0).

Table 6.7 Confusion matrix for the Unknown vs Unknown dataset (completely unseen). The rows represent the true class whereas the columns represent the predicted class (true positives shown in bold in the diagonal). The precision and recall per class are shown in the last two columns.

	Few (c_0)	Some (c_1)	Many (c_2)	Precision	Recall
Few (c_0)	0.86	0.14	0.00	0.83	0.86
Some (c_1)	0.17	0.71	0.12	0.73	0.71
Many (c_2)	0.00	0.13	0.87	0.88	0.87

The classification proportions per class and the precision and recall are shown in the confusion matrix in Table 6.7. As can be seen, the model has most difficulties when classifying the items of the middle class (*i.e.*, c_1), since its borders with other classes are more diffuse. This is a product of the partitioning created by the thresholds, which, although aligned with the three sections of the targets distribution curve (see Figure 6.9), create somewhat of an unnatural separation. The errors around these boundaries are further explored in Figure 6.9, which shows the distribution of targets along the average error rate around the classes.

Figure 6.9 clearly shows that most of the error occurs when classifying instances of the class *some* (central region) which are close to the borders of said class, sometimes being classified as *few* in the lower side or as *many* in the higher side. However, the error accumulated does not represent a threat to the scheduling, even when misclassifying, since a comparison whose number of hits is in the border of two classes will complete without error regardless of which class it was assigned to. On the contrary, high error rates observed some place else than the boundaries of the classes would be a concern since it would most certainly end in either a system crash or wasted resources.

The same experiment is performed from the perspective of a regression problem (rather than classification) to remove the forced boundaries between classes in subsection 6.2.7 "Predictions as a regression problem".

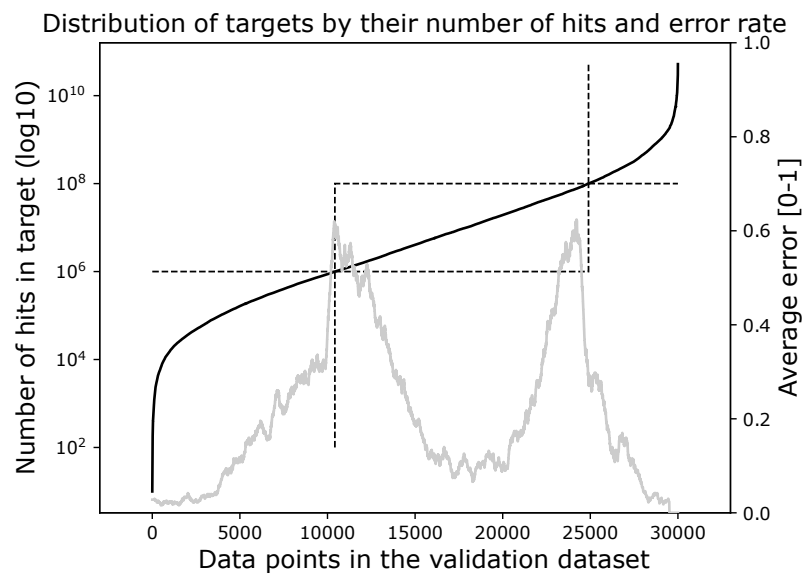


Fig. 6.9 Distribution of validation targets (dark black line) in $\log_{10}$ scale showing the regions belonging to each class (*few*, *some* and *many*, enclosed in dotted lines) as well as the error throughout the classification (shown in light gray in the secondary axis). The x-axis contains each of the samples of the validation dataset sorted by the target (number of hits). The primary vertical axis shows the number of seeds per sample and the secondary vertical axis includes the average classification error.

6.2.5 Supercomputer scheduling simulation

In order to show how the prediction system can be used to improve the performance of dedicated parallel systems, the two trace executions described in subsection 6.1.8 "Scheduling simulation") of comparative genomics jobs were simulated, namely one where jobs are submitted uniformly and another one featuring real submission times from the CIEMAT supercomputer.

Four models were evaluated: (1) an optimal prediction system with a 100% accuracy, (2) the predictions from the neural network, (3) an arbitrary prediction system which selects a random resource class for every job and (4) a naive user model where the most resources possible are requested for every submission.

The throughput per hour for both simulations is depicted in Figure 6.10. As can be seen, the optimal model (which predicts with a 100% accuracy) results in the smallest makespan since all predictions are correct and thus no crash can occur. The optimal model is followed closely by the neural network (which achieves 81% accuracy), then the random model and lastly the naive user model (which always allocates the maximum resources per job).

Interestingly, the random model achieves higher peak throughput during small time windows in the real distribution trace. This happens when the currently-running larger jobs crash because of incorrect class prediction, which enables small jobs to be executed before the larger jobs get relaunched, since there is a small waiting period until the user notices that a job has crashed, modelled with a normal distribution. This is further confirmed by the sudden drops in throughput once the small jobs have executed and the large ones either remain running or have just been launched as enough resources were made available. Note that this behaviour does not happen in the uniform job distribution trace since incorrectly classified jobs that crashed are not immediately replaced by other smaller, batched jobs.

The total makespan along with the throughput values are shown in Table 6.8. Note that the makespan is slightly smaller in the CIEMAT machine when compared to the uniform one. This is not due to better scheduling policies but simply because of how the submission times of the jobs were distributed at the end of the trace.

The resulting average throughput is 15.94, 15.12, 13.50 and 12.97 for the optimal, neural network, random and naive model, respectively. This translates into the neural net achieving up to 94% of the optimal performance, and up to 12% and 16% more throughput than the random and naive models. While the behaviour of the random model seems surprisingly accurate, it is due to the fact that only three classes are considered in this experiment, and therefore the probability of choosing a class that will crash is only $1/3$, which is the same as choosing the optimal class or choosing a class with more resources. Had the study been conducted with more classes or in a regression setup (see subsection 6.2.7 "Predictions as a

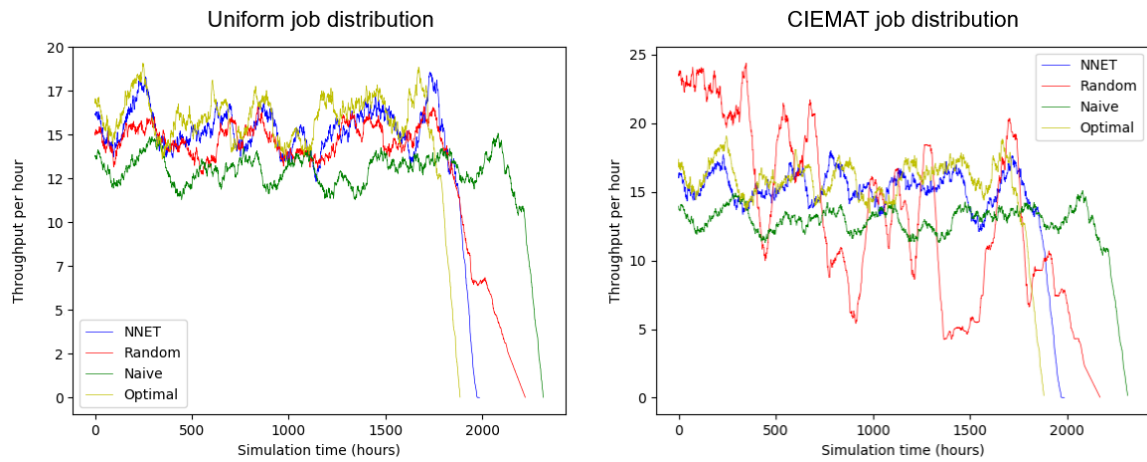


Fig. 6.10 Throughput per hour of the simulation using a uniform job distribution (left) and a real supercomputer job distribution (right). The x-axis corresponds to the time in hours passed from the beginning of the simulation. The y-axis shows the throughput calculated per hour (number of jobs correctly completed).

Table 6.8 Average throughput per hour and total makespan per each employed scheduler in the simulation. Throughput is measured in jobs completed per hour.

	Uniform thr.	Uniform make. (s)	CIEMAT thr.	CIEMAT make. (s)
Optimal	15.90	6,786,092	15.94	6,775,115
Neural network	15.11	7,151,208	15.12	7,133,715
Random	13.81	8,004,801	13.50	7,817,812
Naive	12.96	8,330,913	12.97	8,325,949

regression problem"), the makespan for the random model compared to the neural network and the optimal policies would have been much higher.

Still, further studies should be conducted using different scheduling policies, execution traces and parameters.

Moreover, as explained in Section 6.2.4, an incorrect class assignment from a lower to a higher resource class does not result in a crash but in lesser scheduling efficiency due to higher resource consumption. On the other hand, a lower class prediction incurs in a higher throughput penalty, for not only does it consume time and resources, but also requires an in-between waiting delay and a posterior relaunch.

6.2.6 Explaining predictions via contribution analysis

In this section, the SHapley Additive exPlanations (SHAP) [137] framework is used to explain the performance of the model according to the feature variables. Briefly, the SHAP values represent the significance of each feature variable for the model and their impact in the output prediction. Figure 6.11 shows the mean of the absolute value of the contribution for the eight most and least contributing features.

It can be seen from Figure 6.11 that the first four most contributing trinucleotide words are in fact the same one, either in the query and the reference, and either in its forward or reverse complementary strand (GTA and TAC). The same thing occurs with the following four, namely GTG and CAC and with GGC/GCC. This shows that the method is able to weight words accordingly to their contribution (independently of their representation). Small variations in importance are expected since only 5,000 samples were used to study the contribution (see *e.g.* GGC having slightly more impact than GTG).

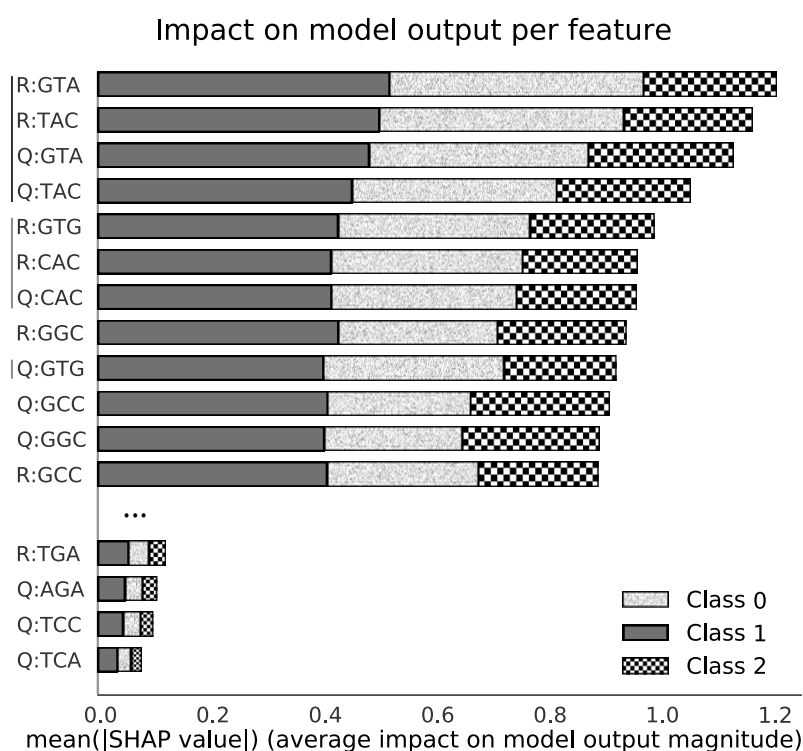


Fig. 6.11 Most and least contributing variables to the classification decision of the model. A Q: represents that the word belongs to the query, whereas a R: represents the reference. Note that originally variables could contribute both positively and negatively to each class.

Still, the huge difference (up to a factor of nearly 20x) in importance of the most and least contributing features is remarkable, and a review on the significance of these motifs

can be found in the literature. On one hand, the motif GTA_n/TAC_n is an alpha satellite repeat commonly found on primates [138]. On the other, the motif CAC_n/GTG_n is one of the most repetitive ones across eukaryotic genomes [139]. Lastly, the motif GGC_n/GCC_n contains the well-known GC box found in the promoter regions of eukaryotic genomes [140].

6.2.7 Predictions as a regression problem

Lastly, by analyzing the errors as a result of the class boundaries, the question is raised of whether the problem should be addressed as a regression rather than a classification. In this line, the last softmax layer in the residual network model was replaced by a dense layer with only one neuron. The model was then retrained (all required datatype conversions were also taken into account) while optimizing the mean squared error. In order to evaluate the result, the difference of magnitudes in $\log_{10}$ scale was used between predictions and targets. Namely:

$$\log_{10} \text{ mean difference} = \sum_{i=0}^n (abs(\log_{10} y_i - \log_{10} \hat{y}_i)) \div n \quad (6.2)$$

Where n is the number of items in the test dataset, y_i are the truth values and $\hat{y}_i$ are the predictions produced by the model. The use of this metric enables to get an estimate of how accurate the model is at predicting the scale of the number of hits in a comparison. Values near 0 imply that the model is able to correctly predict the scale whereas a value of *e.g.* 2 implies that the system mispredicts on average by two magnitude orders (for example, predicting 10^5 seeds when the comparison had actually 10^7 seeds). Given an average $\log_{10}$ difference of less than one, this magnitude-based prediction system could be used to assign the resources of the immediate next magnitude rank, *e.g.* for a prediction of 10^7 seeds, the resources required for a 10^8 would be allocated to the comparison in order to ensure its completion.

Such difference can be interpreted as the attempt to predict the magnitude of the number of hits, *i.e.* similarly to a classification problem with as many classes as orders of magnitude in $\log_{10}$ scale.

The regression model achieved an overall average $\log_{10}$ difference of 0.26 throughout all datasets. A box plot analysis for every dataset is shown in Figure 6.12. Note that the Unknown vs Known dataset was furtherly split into subcategories, see second item below:

1. UvU: Unknown vs Unknown dataset (completely unseen comparison and sequences).

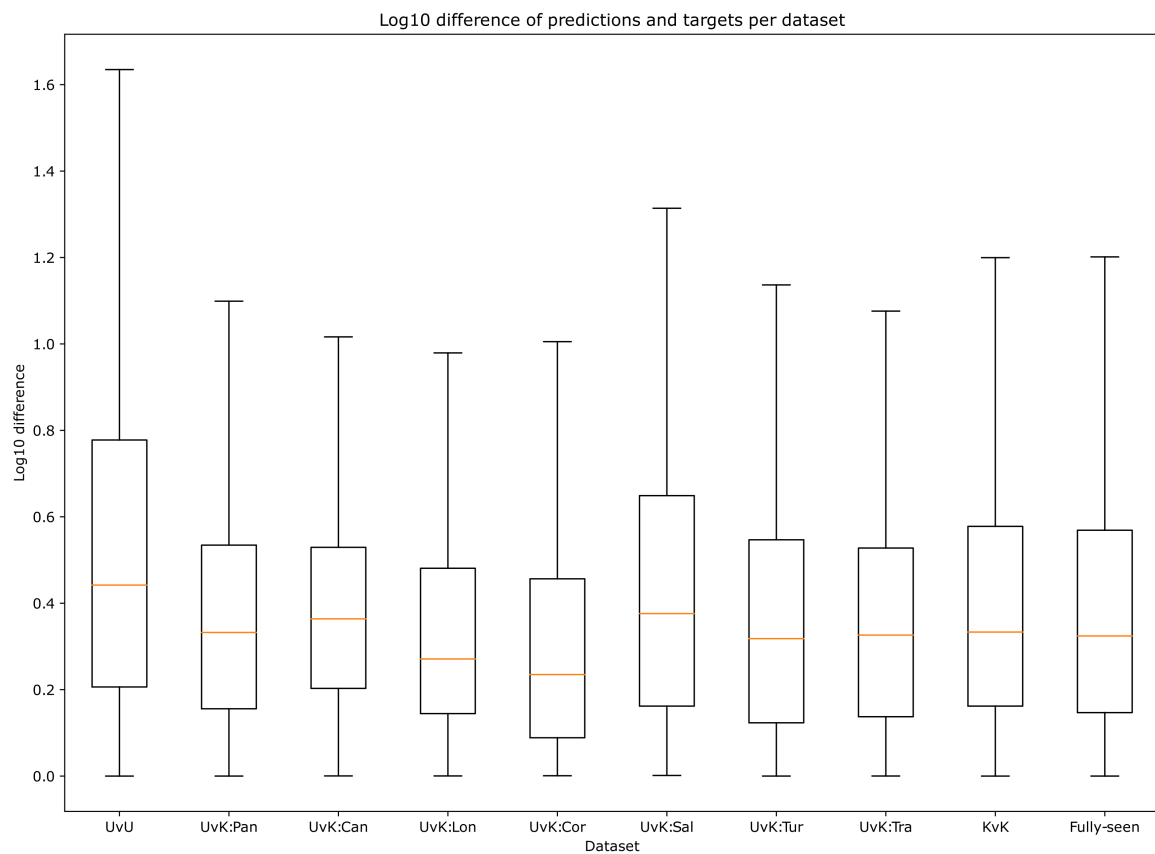


Fig. 6.12 Box plot of the average $\log_{10}$ difference of the predicted and target magnitude per validation dataset.

2. UvK: Unknown vs Known (only the reference sequences were used in the training, namely *Pan troglodytes*, *Canis lupus*, *Lonchura striata*, *Corvus cornix*, *Salmo salar*, *Tursiops truncatus*, *Trachemys scripta elegans*).
3. KvK: Known vs Known (both sequences are known but the target comparison is not).
4. Fully-seen: A subset of the training dataset (both sequences and target are known).

As can be seen from Figure 6.12, all datasets (including the Unknown vs Unknown) achieved an interquartile range below 0.6 difference, with the median being 0.26. Still, some outlier predictions take place in the first quartile of the Unknown vs Unknown dataset which exceed one $\log_{10}$ magnitude difference, nevertheless the proportion of these represent only 7% of the samples (2,303 out of 30,000 samples).

This experiment shows that it would be possible to use the proposed prediction system in a regression scenario where each comparison gets allocated a continuous value of resources, for example number of CPUs, walltime, memory, *etc*, rather than a single, broad class of resources, therefore allowing for a more fine-grained and efficient scheduling policy.

Chapter 7

Conclusions and future work

7.1 Conclusions

In this thesis, four works related to the fields of Computer Architecture, Computer Science and Computational Biology have been presented. These specifically comprise the use of different hardware architectures, including single processors, shared-memory machines and Graphics Processing Units to accelerate the computationally expensive task of sequence alignment, which is central to the fields of Life Sciences, Bioinformatics and Precision Medicine. An additional level of abstraction which joins the three disciplines is also included, namely by studying the improvement in efficiency of schedulers aided by Machine Learning models.

The first work, described in chapter 3 "Ultra-fast sequence comparison on single-processor systems", proposes a lightweight processing algorithm that can run on single cores using a minimal amount of memory with nearly no I/O requirements for the comparison of massive, all vs. all genomic sequences. It combines a probabilistic hashing technique and indexing structure with a mathematical framework to produce a coarse-grained signature representation of the sequence comparison, hence allowing researchers to quickly analyze and explore sequence comparisons with a speedup of over 10x in runtime and nearly 50x less memory consumption. Furthermore, the algorithm is particularly suitable for task-level parallelization, since its low computational requirements enable to run as many instances as cores are available on the target machine. Future parallel improvements are described in the following section 7.2 "Future work".

Chapter 3 of this thesis was elaborated based on the following journal publication:

- **Pérez-Wohlfeil, Esteban**, Sergio Diaz-del-Pino, and Oswaldo Trelles. "Ultra-fast genome comparison for large-scale genomic experiments." *Scientific reports* 9, no. 1 (2019): 1-10. (Q1, IF 3.99).

The second work, addressed in chapter 4 "Shared-memory multiprocessing for the fine-grained alignment of reads", comprises the parallelization of metagenomic comparisons using shared-memory machines. The proposed method includes first heuristic optimizations to the Needleman-Wunsch dynamic programming algorithm to lower the runtime and memory requirements of the optimal sequence alignment problem, and then builds on these optimizations to provide with a parallel implementation that benefits from large multi-core caches which results in superlinear speedup. Coupled with a custom scheduler whose granularity is based on the underlying data type, the proposed method is able to produce higher parallel speedup than other state-of-the-art methods, while also providing optimal and gapped alignments. Moreover, the relevancy of parameters in computational bioinformatics is brought up and discussed, addressing the problems of bias and excessive fine-tuning, and for which a solution is provided.

Chapter 4 of this thesis was elaborated based on the following journal and conference publications:

- **Pérez-Wohlfeil, Esteban**, and Oswaldo Trelles. "Precise and parallel pairwise metagenomic comparisons." *Journal of Computational Biology* 25, no. 8 (2018): 841-849 (Q4, IF 0.89).
- **Pérez-Wohlfeil, Esteban**, Oscar Torreno, and Oswaldo Trelles. "Pairwise and incremental multi-stage alignment of metagenomes: a new proposal." In *International Conference on Bioinformatics and Biomedical Engineering*, pp. 74-80. Springer, Cham, 2017 (International conference).
- **Pérez-Wohlfeil, Esteban**, Jose A. Arjona-Medina, Oscar Torreno, Eugenia Ulzurrun, and Oswaldo Trelles. "Computational workflow for the fine-grained analysis of metagenomic samples." *BMC genomics* 17, no. 8 (2016): 351-361 (Q1, IF 3.72).

The third work, namely the acceleration of irregular and exhaustive sequence comparison (described in chapter 5 "GPU-acceleration for the irregular and exhaustive sequence comparison") includes the successful transformation of each of the stages of the GECKO algorithm into a fully data-parallel method which can be run efficiently in GPUs. This work proposes a batching methodology coupled with several mechanisms to deal with the unpredictable load, including (1) efficient k -mer computation based on cache broadcasting of

constants, (2) block computation of seed matches based on min reduction and their dedicated/synchronized handling of store transactions, (3) thread-cooperative extension of arbitrarily long alignments and (4) a static partitioning of the alignment tasks to avoid redundant computation. The combination of these techniques results in a CUDA GPU implementation which is able to run between 10x to 6x times faster than other state-of-the-art GPU implementations, and between one to two orders of magnitude faster than its CPU counterpart.

Chapter 5 of this thesis was elaborated based on the following article:

- **Pérez-Wohlfeil, Esteban**, Oswaldo Trelles and Nicolas Guil. “Irregular alignment of arbitrarily long DNA sequences on GPU”, *The Journal of Supercomputing* (2022): 1-30. (Q2, IF 2.42)

The last contribution (described in chapter 6 "Scheduling of genome comparisons using Machine Learning") features the use of Machine Learning models to improve the scheduling capability of supercomputers and cloud computing providers by predicting the amount of computational resources required in sequence comparisons. The findings suggest that it could be possible to accurately know in advance the number of common words between sequences, which could be used for different purposes, particularly (1) in combination with schedulers to aid in the dispatching of jobs and (2) as an estimation of similarity between species (without computation nor indexing) and therefore apply it to massive evolutionary studies. The first point is of special interest, since the number of matches roughly translates into the amount of computation required per sequence comparison, and therefore a scheduler could select and allocate automatically the precise amount of resources needed as opposed to a user selection, which has been proven to be inaccurate in many circumstances [121].

Chapter 6 was elaborated based on the following conference publication:

- **Pérez-Wohlfeil, Esteban** and Oswaldo Trelles. “Towards the efficient scheduling of sequence comparisons using Machine Learning”, *Jornadas Sarteco 2021, VI Congreso Nacional de Informática (CEDI)*, Málaga, Spain, Sep 2021 (National conference).

Altogether, the works in this thesis have (1) reduced the computational complexity of performing sequence comparisons, both for coarse and fine-grained experiments; (2) described integral parallelization techniques and mechanisms that incorporate domain-information knowledge to increase speedup, efficiency and output quality and (3) provided researchers with new tools to produce significant research and opened new research paths and questions.

Additionally, it must be pointed out that all the objectives set for this thesis (see section 1.1 "Research objectives") were achieved. From the perspective of Computer Science:

- The study and description of a formal algorithmic theory that supports and unifies each of the architectural and implementation decisions was established in the article corresponding to chapter 3 "Ultra-fast sequence comparison on single-processor systems", where the theoretical foundations were laid out in order to understand the magnitudes of the processes involved in sequence comparison, and therefore what type of processing was suitable and required (*e.g.* coarse *vs.* fine-grained, task *vs.* data parallelism, granularity, *etc.*).
- A review describing the nature of the informational and computational processes that take place in comparative genomics is provided throughout this thesis in section 2.3 "Sequence comparison", where the differences with generic string matching are highlighted and the most prevalent data structures and algorithms are addressed from a computational perspective. Moreover, this is further accompanied by the introductions and background sections of the published articles where, as opposed to a general review, provide a more in-depth state-of-the-art dive for the particular content of the publication.

From the perspective of Computer Architecture:

- The reviewing and understanding of the existing landscape of computer architecture, which was a prerequisite for both development and communication of findings, was not only limited to traditional processors, shared and distributed memory machines and Graphics Processing Units. Additional work was put into deploying a k -mer processing algorithm for Field Programmable Gate Arrays that was used as prototype and proof-of-concept for research proposals.
- The development of new methodologies for the application of both High Performance Computing and parallelization techniques is materialized into the different research publications, which employ single-processors, shared-memory machines and Graphics Processing Units. Furthermore, the last proposed work, which represents another architectural layer, is focused on improving the efficiency of scheduling mechanisms in Supercomputers by applying Machine Learning algorithms.

From the perspective of Computational Biology:

- A reduction of the computational complexity of genome comparison has been achieved for coarse and fine-grained sequence comparisons using single processors as well as accelerators, which translates into researchers having a more powerful and broad range of tools with which to perform comparisons in seconds instead of hours or days.

7.2 Future work

Several new research paths are now open after the studies conducted in this thesis. It is the wish of the author to continue researching in these topics wherever and whenever possible. In particular:

- Inclusion of an efficient, thread-cooperative gap aligner within GPUGECKO. The work presented in section 5.1 "Methods" is a seed-based sequence comparison algorithm. This means that by design, the output alignments include only High-scoring Segment Pairs in the same diagonal and therefore, if a gap (an insertion or deletion) occurs in any of the two sequences involved, the alignment will be incomplete. To overcome such limitation, a gap-extending module can be included, either as a host co-routine using vectorized intrinsics, or directly as a CUDA kernel. In the latter case, it would be interesting to achieve a similar level of thread cooperation as in the ungapped extension kernel featured in section 5.1.6 "Extension of seeds". This is certainly one of the aims of the author as it would allow for a much broader application range.
- Continue and extend the study on parameterization on sequence alignment experiments as featured in section 4.1 "Methods". In particular, the presented work suggested that it is possible to estimate the boundaries between errors, noise and bias within a sequence comparison by analyzing the parameters and the outcome of the comparisons (*i.e.*, including both data and algorithmic settings). By extending on a formal theory, it could be possible to prove the optimality of certain parameters which eventually would lead to a reduction in computational cost.
- Using Machine Learning to predict the number of large seeds between two genomes based on the frequencies of smaller k -words. As described throughout this thesis, the use of small k -words for genome comparison is extremely fast since no indexing data structure is required. However, it results in huge amounts of noise and therefore its application is limited. Promising results have been achieved by using these small words to train a neural network in order to predict the amount of seeds of larger size, hence enabling supercomputers and cloud providers to make a more efficient scheduling of computational resources when running sequence comparisons. The inclusion of further genomic sequences to make datasets more consistent can potentially reduce variance and improve the overall accuracy of all the methods, while also allowing to split the dataset into training, validation and testing in order to obtain a more realistic generalization accuracy. Further and thorough testing of different schedulers (*e.g.* using priority queues, backfilling, I/O aware algorithms, *etc.*) and the inclusion of

more precise runtimes, memory consumption and CPU usage values would result in a better understanding of the efficiency improvement given a machine-learning aided scheduler. Lastly, it would also be of interest to explore further architectures for the neural network as well as alter the structure of the proposed one.

- Design a primary-secondary interface to optimize the distributed shuffling of the probabilistically indexed hash dictionary in subsection 3.1.4 "Indexing" such that in all vs. all comparisons the index is calculated only once. This would produce significant acceleration since (1) indices would only need to be computed once per sequence and (2) I/O reads from secondary storage would be greatly reduced while replacing them with small network-distributed transfers. However, due to the low running times, a trade-off study must be performed since passively waiting for the indices could increase makespan if the sequences are small. Additionally, speculative computation could also be employed.
- Fine-tune the genomic distance metric proposed in subsection 3.1.7 "Scoring of comparisons". The suggested metric combines information related to the coverage of the sequences and the number of evolutionary events. Nevertheless, it is too coarse-grained for some particular cases and lacks the incorporation of positional or structural information. A more refined metric would allow a better scoring of sequence comparisons and become a standard within the field of comparative genomics.

7.3 Additional work

7.3.1 Additional conferences and publications

Further scientific publications which are not directly supporting this thesis have also been produced, either as first author or co-author. These include national and international conferences and publications with international collaborators:

Conferences:

- **Pérez-Wohlfeil, Esteban**, Chicano, Francisco, Alba, Enrique. An Intelligent Data Analysis of the Structure of NP Problems for Efficient Solution: The Vehicle Routing Case. Proceedings of the Fourth Euro-China Conference on Intelligent Data Analysis and Applications. ECC 2017. Advances in Intelligent Systems and Computing, vol 682 (2018). Springer, Cham.
- Rodríguez-Brazzarola, Pablo, **Pérez-Wohlfeil, Esteban**, Díaz-del-Pino, Sergio, Holthausen, Ricardo, Trelles, Oswaldo. "Analyzing the Differences Between Reads and Contigs

When Performing a Taxonomic Assignment Comparison in Metagenomics." *Bioinformatics and Biomedical Engineering*. IWBBIO 2018. Lecture Notes in Computer Science, vol 10813 (2018). Springer, Cham.

- Holthausen, Ricardo, Díaz-Del-Pino, Sergio, **Pérez-Wohlfeil, Esteban**, Rodríguez-Brazzarola, Pablo., Trelles, Oswaldo. "Workflows and Service Discovery: A Mobile Device Approach". *Bioinformatics and Biomedical Engineering*. IWBBIO 2018. Lecture Notes in Computer Science, vol 10813 (2018). Springer, Cham.

Journal publications:

- Siqueira, Franciele Maboni, **Esteban Pérez-Wohlfeil**, Fabíola Marques Carvalho, Oswaldo Trelles, Irene Silveira Schrank, Ana Tereza Ribeiro Vasconcelos, and Arnaldo Zaha. "Microbiome overview in swine lungs." *PLoS One* 12, no. 7 (2017): e0181503 (Q1, IF 2.76).
- **Pérez-Wohlfeil, Esteban**, Oscar Torreno, Louisa J. Bellis, Pedro L. Fernandes, Brane Leskosek, and Oswaldo Trelles. "Training bioinformaticians in high performance computing." *Heliyon* 4, no. 12 (2018): e01057 (Q2, 3.35).
- Diaz-del-Pino, Sergio, Rodriguez-Brazzarola, Pablo, **Pérez-Wohlfeil, Esteban**, Trelles Oswaldo. "Combining Strengths for Multi-genome Visual Analytics Comparison". *Bioinformatics and Biology Insights*. 2019;13 (Q4, 0.46).
- Ulzurrun, Eugenia, Duarte, Yorley, **Pérez-Wohlfeil, Esteban**, Gonzalez-Nilo, Fernando, Trelles, Oswaldo. "PLIDflow: an open-source workflow for the online analysis of protein–ligand docking using galaxy". *Bioinformatics*, Volume 36, Issue 14, 15 July 2020, Pages 4203–4205 (Q1, IF 1.96).
- Diaz-del-Pino, Sergio, **érez-Wohlfeil, Esteban**, and Oswaldo Trelles. "Unraveling Genome Evolution Throughout Visual Analysis: The XCut Portal." *Bioinformatics and Biology Insights* 15 (2021): 11779322211021422 (Q2, IF 0.84).

In particular, it is worth a short mention regarding the article *Training bioinformaticians in high performance computing*, which featured an international collaboration between colleagues from the European ELIXIR framework for Life-Sciences research. The article focuses on overcoming the existing barriers and difficulties of teaching High Performance Computing to researchers from other fields, with particular emphasis in Computational biologists.

7.3.2 Workshops

The presented doctoral studies have also been validated through several training events and workshops in both the contexts of Computer Science and Computational Biology. Regarding international workshops, three were deployed as main speaker, an additional one as a co-speaker and lastly an online one (due to the COVID-19 pandemic). These are:

- As co-speaker and technical assistant, the ELIXIR-EXCELERATE High Performance Computing "Train-the-Researcher" course in April 2017 in Málaga.
- The 2-hour workshop session "Advanced Computing for Comparative Genomics" in the PRACE ("Partnership for Advanced Computing in Europe") Scientific and Industrial Conference (PRACEdays18) at the University of Ljubljana (Slovenia), Faculty of Mechanical Engineering in May 2018.
- The 6 and 3 hours workshop at the "Microbiome Health Workshop" at the Faculty of Sciences Dhar Mahraz (Morocco), editions of 2017 and 2018.
- The fully-online workshop "From small to large-scale genome comparison" at the European Galaxy Training portal, part of the ELIXIR European project for Life Sciences in January 2021.

7.3.3 International research stay

As part of the international doctoral studies, the author carried out a 3-month research stay at the Institute for Machine Learning at the Johannes Kepler University (Linz, Austria) under the supervision of Prof. Sepp Hochreiter, which began the 16th of September and came to an end on the 16th of December of 2021. Throughout this stay the author expanded his knowledge in regards to Machine Learning, particularly in (1) its application to time series using Long-Short Term Memory neurons (invented by Prof. Sepp Hochreiter), (2) modern recent models such as transformers for temporal data and (3) tabular data networks such as SAINT.

References

- [1] David Schneider. The exascale era is upon us: The frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second. *IEEE Spectrum*, 59(1):34–35, 2022.
- [2] Zdzislaw Polkowski and Malgorzata Nycz-Lukaszewska. Infrastructure as a service: Cloud computing model for micro companies. In *Advances in Data Science and Management*, pages 307–320. Springer, 2020.
- [3] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.
- [4] Lizhe Wang, Yan Ma, Jining Yan, Victor Chang, and Albert Y Zomaya. pipscloud: High performance cloud computing for remote sensing big data management and processing. *Future Generation Computer Systems*, 78:353–368, 2018.
- [5] Victor Chang. Towards data analysis for weather cloud computing. *Knowledge-Based Systems*, 127:29–45, 2017.
- [6] Joseph Emeras, Sebastien Varrette, Valentin Plugaru, and Pascal Bouvry. Amazon elastic compute cloud (ec2) versus in-house hpc platform: A cost analysis. *IEEE Transactions on Cloud Computing*, 7(2):456–468, 2016.
- [7] Adam G Carlyle, Stephen L Harrell, and Preston M Smith. Cost-effective hpc: The community or the cloud? In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 169–176. IEEE, 2010.
- [8] Tianyi David Han and Tarek S Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed systems*, 22(1):78–90, 2010.
- [9] A Capozzoli, O Kilic, C Curcio, and A Liseno. The success of gpu computing in applied electromagnetics. *The Applied Computational Electromagnetics Society Journal (ACES)*, pages 148–151, 2018.
- [10] Andreas D Baxevanis, Gary D Bader, and David S Wishart. *Bioinformatics*. John Wiley & Sons, 2020.
- [11] Philip Hieter and Mark Boguski. Functional genomics: it’s all how you read it. *Science*, 278(5338):601–602, 1997.
- [12] Thomas C Terwilliger, David Stuart, and Shigeyuki Yokoyama. Lessons from structural genomics. *Annual review of biophysics*, 38:371–383, 2009.

- [13] John P Huelsenbeck, Bruce Rannala, and John P Masly. Accommodating phylogenetic uncertainty in evolutionary studies. *Science*, 288(5475):2349–2350, 2000.
- [14] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.
- [15] William Aspray. John von neumann’s contributions to computing and computer science. *Annals of the History of Computing*, 11(3):189–195, 1989.
- [16] HT Kung. Ce leiseron," systolic arrays for vlsi. In *Proc. Symp. Sparse Matrix Computations and Their Applications*, number 2-3, pages 256–282, 1978.
- [17] David B Skillicorn. A taxonomy for computer architectures. *Computer*, 21(11):46–57, 1988.
- [18] Christoph Lameter. Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51, 2013.
- [19] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open mpi: A flexible high performance mpi. In *International Conference on Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2005.
- [20] C Darwin. The origin of species (new york: Pf collier), 1909.
- [21] BS Strauss. Dna repair mechanisms and their relation to mutation and recombination. Technical report, Univ. of Chicago, 1968.
- [22] Michele Cargill, David Altshuler, James Ireland, Pamela Sklar, Kristin Ardlie, Nila Patil, Charles R Lane, Esther P Lim, Nilesh Kalyanaraman, James Nemesh, et al. Characterization of single-nucleotide polymorphisms in coding regions of human genes. *Nature genetics*, 22(3):231–238, 1999.
- [23] Aya Jokobovits and Hirohisa Tsuda. Generation of large genomic dna deletions, December 7 1999. US Patent 5,998,209.
- [24] Nilesh K Banavali. Partial base flipping is sufficient for strand slippage near dna duplex termini. *Journal of the American Chemical Society*, 135(22):8274–8282, 2013.
- [25] C-K Peng, Sergey V Buldyrev, Shlomo Havlin, Michael Simons, H Eugene Stanley, and Ary L Goldberger. Mosaic organization of dna nucleotides. *Physical review e*, 49(2):1685, 1994.
- [26] Iupac code. <https://www.bioinformatics.org/sms/iupac.html> (last checked: 02-02-2021).
- [27] Julie D Allen, Deborah Anderson, Joe Becker, Richard Cook, Mark Davis, Peter Edberg, Michael Everson, Asmus Freytag, Laurentiu Iancu, Richard Ishida, et al. The unicode standard. *Mountain view, CA*, pages 660–664, 2012.
- [28] Martin Farach and S Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Annual Symposium on Combinatorial Pattern Matching*, pages 130–140. Springer, 1996.

- [29] Mehryar Mohri. String-matching with automata. *Nord. J. Comput.*, 4(2):217–231, 1997.
- [30] Benjamin C Brodie, David E Taylor, and Ron K Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *ACM SIGARCH computer architecture news*, 34(2):191–202, 2006.
- [31] Alessandro Comodi, Davide Conficconi, Alberto Scolari, and Marco D Santambrogio. Tired: Tiled regular expression matching architecture. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 131–137. IEEE, 2018.
- [32] ES James. A study of branch prediction strategies, in 25 years of the international symposia on computer architecture (selected papers). In *ISCA'98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, 1998.
- [33] Pei Cao, Edward W Felten, Anna R Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *ACM SIGMETRICS Performance Evaluation Review*, 23(1):188–197, 1995.
- [34] Gonzalo Navarro. Nr-grep: a fast and flexible pattern-matching tool. *Software: Practice and Experience*, 31(13):1265–1312, 2001.
- [35] Yue Huang and Ling Zhang. Rapid and sensitive dot-matrix methods for genome analysis. *Bioinformatics*, 20(4):460–466, 2004.
- [36] Hilde Vinje, Kristian Hovde Liland, Trygve Almøy, and Lars Snipen. Comparing k-mer based methods for improved classification of 16s sequences. *BMC bioinformatics*, 16(1):1–13, 2015.
- [37] Benny Chor, David Horn, Nick Goldman, Yaron Levy, and Tim Massingham. Genomic dna k-mer spectra: models and modalities. *Genome biology*, 10(10):1–10, 2009.
- [38] Kai Lai Chung. Markov chains. *Springer-Verlag, New York*, 1967.
- [39] Qi Dai, Yanchun Yang, and Tianming Wang. Markov model plus k-word distributions: a synergy that produces novel statistical measures for sequence comparison. *Bioinformatics*, 24(20):2296–2302, 2008.
- [40] Huili Liu. A joint probabilistic model in dna sequences. *Current Bioinformatics*, 13(3):234–240, 2018.
- [41] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.
- [42] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.
- [43] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

- [44] Asiya Nazir. Review on metagenomics and its applications. *Imp J Intersd Res*, 2(10), 2016.
- [45] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.
- [46] Jessica Alföldi and Kerstin Lindblad-Toh. Comparative genomics as a tool to understand evolution and disease. *Genome research*, 23(7):1063–1068, 2013.
- [47] Scott McGinnis and Thomas L Madden. Blast: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic acids research*, 32(suppl_2):W20–W25, 2004.
- [48] Esteban Pérez-Wohlfeil, Sergio Diaz-del Pino, and Oswaldo Trelles. Ultra-fast genome comparison for large-scale genomic experiments. *Scientific reports*, 9(1):1–10, 2019.
- [49] Pall Melsted and Jonathan K Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC bioinformatics*, 12(1):1–7, 2011.
- [50] Stephen F Altschul, Mark S Boguski, Warren Gish, and John C Wootton. Issues in searching molecular sequence databases. *Nature genetics*, 6(2):119–129, 1994.
- [51] DRVLB Thambawita, Roshan G Ragel, and Dhammike Elkaduwe. An optimized parallel failure-less aho-corasick algorithm for dna sequence matching. In *2016 IEEE International Conference on Information and Automation for Sustainability (ICIAfS)*, pages 1–6. IEEE, 2016.
- [52] Arthur L Delcher, Steven L Salzberg, and Adam M Phillippy. Using mummer to identify similar regions in large sequence sets. *Current protocols in bioinformatics*, (1):10–3, 2003.
- [53] Oscar Torreno and Oswaldo Trelles. Breaking the computational barriers of pairwise genome comparison. *BMC bioinformatics*, 16(1):1–13, 2015.
- [54] Jared T Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26(12):i367–i373, 2010.
- [55] Ross A Lippert. Space-efficient whole genome comparisons with burrows–wheeler transforms. *Journal of computational biology*, 12(4):407–415, 2005.
- [56] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357–359, 2012.
- [57] Danilo Bzdok, Martin Krzywinski, and Naomi Altman. Machine learning: a primer. *Nature methods*, 14(12):1119, 2017.
- [58] Rishabh Choudhary and Hemant Kumar Gianey. Comprehensive review on supervised machine learning algorithms. In *2017 International Conference on Machine Learning and Data Science (MLDS)*, pages 37–43. IEEE, 2017.
- [59] Raul Rojas. The backpropagation algorithm. In *Neural networks*, pages 149–182. Springer, 1996.

- [60] Zoubin Ghahramani. Unsupervised learning. In *Summer school on machine learning*, pages 72–112. Springer, 2003.
- [61] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Joelle Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.
- [62] Sean D Holcomb, William K Porter, Shaun V Ault, Guifen Mao, and Jin Wang. Overview on deepmind and its alphago zero ai. In *Proceedings of the 2018 international conference on big data and education*, pages 67–71, 2018.
- [63] Carl Kingsford and Steven L Salzberg. What are decision trees? *Nature biotechnology*, 26(9):1011–1013, 2008.
- [64] Alexey Natekin and Alois Knoll. Gradient boosting machines, a tutorial. *Frontiers in neurorobotics*, 7:21, 2013.
- [65] Wojciech Samek, Grégoire Montavon, Sebastian Lapuschkin, Christopher J Anders, and Klaus-Robert Müller. Explaining deep neural networks and beyond: A review of methods and applications. *Proceedings of the IEEE*, 109(3):247–278, 2021.
- [66] Laura Elena Raileanu and Kilian Stofel. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93, 2004.
- [67] Jehad Ali, Rehanullah Khan, Nasir Ahmad, and Imran Maqsood. Random forests and decision trees. *International Journal of Computer Science Issues (IJCSI)*, 9(5):272, 2012.
- [68] Simon Bernard, Laurent Heutte, and Sebastien Adam. On the selection of decision trees in random forests. In *2009 International Joint Conference on Neural Networks*, pages 302–307. IEEE, 2009.
- [69] Andrei V Konstantinov and Lev V Utkin. Interpretable machine learning with an ensemble of gradient boosting machines. *Knowledge-Based Systems*, 222:106993, 2021.
- [70] Kenji Suzuki. *Artificial neural networks: Architectures and applications*. BoD–Books on Demand, 2013.
- [71] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 2021.
- [72] Omer Berat Sezer, Mehmet Ugur Gudelek, and Ahmet Murat Ozbayoglu. Financial time series forecasting with deep learning: A systematic literature review: 2005–2019. *Applied soft computing*, 90:106181, 2020.
- [73] Robert Dale. Gpt-3: What’s it good for? *Natural Language Engineering*, 27(1):113–118, 2021.

- [74] Jinghui Chen, Saket Sathe, Charu Aggarwal, and Deepak Turaga. Outlier detection with autoencoder ensembles. In *Proceedings of the 2017 SIAM international conference on data mining*, pages 90–98. SIAM, 2017.
- [75] Jelena Kocić, Nenad Jovičić, and Vujo Drndarević. An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms. *Sensors*, 19(9):2064, 2019.
- [76] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.
- [77] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *Journal of machine learning research*, 10(1), 2009.
- [78] Diogo V Carvalho, Eduardo M Pereira, and Jaime S Cardoso. Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8(8):832, 2019.
- [79] Philippe Aghion, Benjamin F Jones, and Charles I Jones. Artificial intelligence and economic growth. In *The economics of artificial intelligence: An agenda*, pages 237–282. University of Chicago Press, 2018.
- [80] Sara Goodwin, John D McPherson, and W Richard McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, 17(6):333–351, 2016.
- [81] Paul VC Hough. Method and means for recognizing complex patterns, December 18 1962. US Patent 3,069,654.
- [82] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.
- [83] Jose Antonio Arjona Medina et al. Algorithms and methods for large-scale genome rearrangements identification. 2017.
- [84] Ryuichiro Nakato and Osamu Gotoh. Cgaln: fast and space-efficient whole-genome alignment. *BMC bioinformatics*, 11(1):1–14, 2010.
- [85] Guillaume Marçais, Arthur L Delcher, Adam M Phillippy, Rachel Coston, Steven L Salzberg, and Aleksey Zimin. Mummer4: A fast and versatile genome alignment system. *PLoS computational biology*, 14(1):e1005944, 2018.
- [86] Ying Chen, Weicai Ye, Yongdong Zhang, and Yuesheng Xu. High speed blastn: an accelerated megablast search tool. *Nucleic acids research*, 43(16):7762–7768, 2015.
- [87] E Lerat. Identifying repeats and transposable elements in sequenced genomes: how to find your way through the dense forest of programs. *Heredity*, 104(6):520–533, 2010.
- [88] Esteban Pérez-Wohlfeil and Oswaldo Trelles. Precise and parallel pairwise metagenomic comparisons. *Journal of Computational Biology*, 25(8):841–849, 2018.

- [89] Jose Antonio Arjona-Medina, Oscar Torreno, Noura Chelbat, and Oswaldo Trelles. Experimental study of local alignment distributions in the comparison of large genomic sequences. *Sociedad Iberoamericana de Bioinformática*, 2013.
- [90] Jack Bresenham. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM*, 20(2):100–106, 1977.
- [91] Constantine D Polychronopoulos and David J Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *Ieee transactions on computers*, 100(12):1425–1439, 1987.
- [92] Peter J Turnbaugh, Ruth E Ley, Micah Hamady, Claire M Fraser-Liggett, Rob Knight, and Jeffrey I Gordon. The human microbiome project. *Nature*, 449(7164):804–810, 2007.
- [93] Peter J Turnbaugh and Jeffrey I Gordon. The core gut microbiome, energy balance and obesity. *The Journal of physiology*, 587(17):4153–4158, 2009.
- [94] Jack Doweck. White paper inside intel® core™ microarchitecture and smart memory access. *Intel Corporation*, 52:72–87, 2006.
- [95] Nicolas Maillet, Guillaume Collet, Thomas Vannier, Dominique Lavenier, and Pierre Peterlongo. Commet: comparing and combining multiple metagenomic datasets. In *2014 IEEE international conference on bioinformatics and biomedicine (BIBM)*, pages 94–98. IEEE, 2014.
- [96] Gaëtan Benoit. Simka: fast kmer-based method for estimating the similarity between numerous metagenomic datasets. In *RCAM*, 2015.
- [97] Yuriy Fofanov, Yi Luo, Charles Katili, Jim Wang, Yuri Belosludtsev, Thomas Powdrill, Chetan Belapurkar, Viacheslav Fofanov, Tong-Bin Li, Sergey Chumakov, et al. How independent are the appearances of n-mers in different genomes? *Bioinformatics*, 20(15):2421–2428, 2004.
- [98] Esteban Pérez-Wohlfeil, Oscar Torreno, and Oswaldo Trelles. Pairwise and incremental multi-stage alignment of metagenomes: a new proposal. In *International Conference on Bioinformatics and Biomedical Engineering*, pages 74–80. Springer, 2017.
- [99] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.
- [100] Jack C Yue and Murray K Clayton. A similarity measure based on species proportions. *Communications in Statistics-theory and Methods*, 34(11):2123–2131, 2005.
- [101] Esteban Perez-Wohlfeil, Oswaldo Trelles, and Nicolás Guil. Irregular alignment of arbitrarily long dna sequences on gpu. *The Journal of Supercomputing*, pages 1–30, 2022.
- [102] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

- [103] Dharendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. Survey of gpu based sorting algorithms. *International Journal of Parallel Programming*, 46(6):1017–1034, 2018.
- [104] John L Gustafson. Reevaluating amdahl’s law. *Communications of the ACM*, 31(5):532–533, 1988.
- [105] Kohji Okamura, John Wei, and Stephen W Scherer. Evolutionary implications of inversions that have caused intra-strand parity in dna. *BMC genomics*, 8(1):1–7, 2007.
- [106] Selim G Akl. *Parallel sorting algorithms*, volume 12. Academic press, 2014.
- [107] Dmitri I Arkhipov, Di Wu, Keqin Li, and Amelia C Regan. Sorting with gpus: A survey. *arXiv preprint arXiv:1709.02520*, 2017.
- [108] Emanuele Manca, Andrea Manconi, Alessandro Orro, Giuliano Armano, and Luciano Milanesi. Cuda-quicksort: an improved gpu-based implementation of quicksort. *Concurrency and computation: practice and experience*, 28(1):21–43, 2016.
- [109] Darko Bozidar and Tomaz Dobravec. Comparison of parallel sorting algorithms. *arXiv preprint arXiv:1511.03404*, 2015.
- [110] Official modern gpu repository. <https://github.com/moderngpu/moderngpu>. (revision 2b39855 on master branch).
- [111] Gianni Franceschini, Shan Muthukrishnan, and Mihai Pătraşcu. Radix sorting with no extra space. In *European Symposium on Algorithms*, pages 194–205. Springer, 2007.
- [112] J Kuzio, R Tatusov, and DJ Lipman. Dust. *Unpublished but briefly described in: Morgulis A, Gertz EM, Schäffer AA, Agarwala R. A Fast and Symmetric DUST Implementation to Mask Low-Complexity DNA Sequences. Journal of Computational Biology*, 13(5):1028–1040, 2006.
- [113] Shibdas Bandyopadhyay and Sartaj Sahni. Grs—gpu radix sort for multifold records. In *2010 International Conference on High Performance Computing*, pages 1–10. IEEE, 2010.
- [114] I Schubert and JL Oud. There is an upper limit of chromosome size for normal development of an organism. *Cell*, 88(4):515–520, 1997.
- [115] Kaiyong Zhao and Xiaowen Chu. G-blastn: accelerating nucleotide alignment by graphics processors. *Bioinformatics*, 30(10):1384–1391, 2014.
- [116] Martin Svedin, Steven WD Chien, Gibson Chikafa, Niclas Jansson, and Artur Podobas. Benchmarking the nvidia gpu lineage: From early k80 to modern a100 with asynchronous memory transfers. In *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, pages 1–6, 2021.
- [117] Hennadiy Leontyev and James H Anderson. Tardiness bounds for fifo scheduling on multiprocessors. In *19th Euromicro Conference on Real-Time Systems (ECRTS’07)*, pages 71–71. IEEE, 2007.

- [118] Rasmus V Rasmussen and Michael A Trick. Round robin scheduling—a survey. *European Journal of Operational Research*, 188(3):617–636, 2008.
- [119] Wei Tang, Narayan Desai, Daniel Buettner, and Zhiling Lan. Job scheduling with adjusted runtime estimates on production supercomputers. *Journal of Parallel and Distributed Computing*, 73(7):926–938, 2013.
- [120] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and Ponnuswamy Sadayappan. Selective reservation strategies for backfill job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 55–71. Springer, 2002.
- [121] Wei Tang, Narayan Desai, Daniel Buettner, and Zhiling Lan. Analyzing and adjusting user runtime estimates to improve job scheduling on the blue gene/p. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2010.
- [122] Ross A Lippert, Haiyan Huang, and Michael S Waterman. Distributional regimes for the number of k-word matches between two random sequences. *Proceedings of the National Academy of Sciences*, 99(22):13980–13989, 2002.
- [123] Sophie Röhling, Alexander Linne, Jendrik Schellhorn, Morteza Hosseini, Thomas Dencker, and Burkhard Morgenstern. The number of k-mer matches between two dna sequences as a function of k and applications to estimate phylogenetic distances. *Plos one*, 15(2):e0228070, 2020.
- [124] Mark Johnson, Irena Zaretskaya, Yan Raytselis, Yuri Merezuk, Scott McGinnis, and Thomas L Madden. Ncbi blast: a better web interface. *Nucleic acids research*, 36(suppl_2):W5–W9, 2008.
- [125] Weizhong Li, Andrew Cowley, Mahmut Uludag, Tamer Gur, Hamish McWilliam, Silvano Squizzato, Young Mi Park, Nicola Buso, and Rodrigo Lopez. The embl-e-bi bioinformatics web and programmatic tools framework. *Nucleic acids research*, 43(W1):W580–W584, 2015.
- [126] Jianqiu Ge, Jintao Meng, Ning Guo, Yanjie Wei, Pavan Balaji, and Shengzhong Feng. Counting kmers for biological sequences at large scale. *Interdisciplinary Sciences: Computational Life Sciences*, 12(1):99–108, 2020.
- [127] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.
- [128] Donglin Liu and Gautam B Singh. Profile based methods for genomic sequence retrieval. *ACM SIGBIO Newsletter*, 20(3):6–13, 2000.
- [129] Jun Zheng, Furao Shen, Hongjun Fan, and Jinxi Zhao. An online incremental learning support vector machine for large-scale data. *Neural Computing and Applications*, 22(5):1023–1035, 2013.
- [130] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

- [131] Dongwei Chen, Fei Hu, Guokui Nian, and Tiantian Yang. Deep residual learning for nonlinear regression. *Entropy*, 22(2):193, 2020.
- [132] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? *arXiv preprint arXiv:1805.11604*, 2018.
- [133] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.
- [134] Zijun Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–2. IEEE, 2018.
- [135] Manuel Rodríguez-Pascual, Antonio J Rubio-Montero, José A Moríñigo, and Rafael Mayo-García. Execution data logs of a supercomputer workload over its extended lifetime. *Data in brief*, 28:105006, 2020.
- [136] Juan D Rodriguez, Aritz Perez, and Jose A Lozano. Sensitivity analysis of k-fold cross validation in prediction error estimation. *IEEE transactions on pattern analysis and machine intelligence*, 32(3):569–575, 2009.
- [137] Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *arXiv preprint arXiv:1705.07874*, 2017.
- [138] Isidoro Feliciello, Željka Pezer, Dušan Kordiš, Branka Bruvo Mađarić, and Đurđica Ugarković. Evolutionary history of alpha satellite dna repeats dispersed within human genome euchromatin. *Genome biology and evolution*, 12(11):2125–2138, 2020.
- [139] C Epplen, G Melmer, I Siedlaczek, F-W Schwaiger, W Mäueler, and JT Epplen. On the essence of “meaningless” simple repetitive dna in eukaryote genomes. *DNA fingerprinting: State of the Science*, pages 29–45, 1993.
- [140] MARIA Lundin, Jan Olof Nehlin, and Hans Ronne. Importance of a flanking at-rich region in target site recognition by the gc box-binding zinc finger protein mig1. *Molecular and cellular biology*, 14(3):1979–1985, 1994.
- [141] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11:2007, 2007.
- [142] Chris McClanahan. History and evolution of gpu architecture. *A Survey Paper*, 9, 2010.
- [143] Nathan Otterness and James H Anderson. Amd gpus as an alternative to nvidia for supporting real-time workloads. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- [144] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous branch and warp interweaving for sustained gpu performance. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 49–60. IEEE, 2012.
- [145] ThienLuan Ho, Seung-Rohk Oh, and HyunJin Kim. A parallel approximate string matching under levenshtein distance on graphics processing units using warp-shuffle operations. *PloS one*, 12(10):e0186251, 2017.

- [146] Jie Wang, Xinfeng Xie, and Jason Cong. Communication optimization on gpu: A case study of sequence alignment algorithms. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 72–81. IEEE, 2017.
- [147] Wai-Kong Lee, Hon-Sang Cheong, Raphael C-W Phan, and Bok-Min Goi. Fast implementation of block ciphers and prngs in maxwell gpu architecture. *Cluster Computing*, 19(1):335–347, 2016.
- [148] Anumeena Sorna, Xiaohe Cheng, Eduardo D’azevedo, Kwai Won, and Stanimire Tomov. Optimizing the fast fourier transform using mixed precision on tensor core hardware. In *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, pages 3–7. IEEE, 2018.
- [149] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*, pages 46–57, 2019.
- [150] Scott E Thompson and Srivatsan Parthasarathy. Moore’s law: the future of si micro-electronics. *Materials today*, 9(6):20–25, 2006.
- [151] Tekla S Perry. Move over, moore’s law. make way for huang’s law [spectral lines]. *IEEE Spectrum*, 55(5):7–7, 2018.
- [152] Sanjay Churiwala and I Hyderabad. Designing with xilinx® fpgas. In *Circuits & Systems*. Springer, 2017.
- [153] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.
- [154] Elias Ahmed and Jonathan Rose. The effect of lut and cluster size on deep-submicron fpga performance and density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):288–298, 2004.
- [155] Ding Guangxin, Chen Lingdou, and Liu Zhongli. Design for an io block array in a tile-based fpga. *Journal of Semiconductors*, 30(8):085008, 2009.
- [156] Mohammed AS Khalid and Jonathan Rose. *The effect of fixed I/O pin positioning on the routability and speed of FPGAs*. University of Toronto. Computer Systems Research Institute, 1995.
- [157] Lesley Shannon, Veronica Cojocar, Cong Nguyen Dao, and Philip HW Leong. Technology scaling in fpgas: Trends in applications and architectures. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 1–8. IEEE, 2015.

Appendix A

Hardware differences for exploiting performance in CPUs, GPUs and FPGAs

This Appendix gives an overview of the main components of traditional CPUs, GPUs (specifically the architecture from NVIDIA) and FPGAs.

A.1 Central Processing Unit (CPU)

In essence, a CPU is a specific electronic circuitry designed to perform a set of operations, normally arithmetic, input and output, logic and control operations. In comparison, a GPU is a specialized processor composed of large arrays of cores (notice that modern CPUs also have a few of these) with a slightly different memory architecture and instruction set. On the other side of the spectrum, an FPGA differs from these in the sense that they are not processors but reprogrammable devices which can too be transformed into some type of processors.

While the term processor has been around since the early 1950s, their design has not changed radically ever since. Common processors still follow the Von Neumann's architecture (although extended greatly with additional modules and techniques) which basically employs an arithmetic and logic unit (known as ALU), a set of registers where data is supplied in order to be operated on and a control unit that dictates which operation must be performed by fetching it from an instruction memory where the program to be executed is allocated. Therefore its functioning proceeds by (1) fetching instructions from the program memory (with additional data that can be transferred from external and typically much larger memory spaces), (2) decoding and executing the operation instruction to the operands (which is performed within the ALU and can be *e.g.* a sum or a control flow jump), (3) returning the

result of the operation to the external memories in order to preserve the computation and (4) increasing the program counter to fetch the next instruction. This continues until all instructions have been exhausted and therefore the program is completed.

A.1.1 Architecture overview

In Figure A.1, a coarse-grain, MIPS-like representation of a CPU is shown (therefore no pipelining is included) featuring two cores. First, the control unit is shown including the elements required to perform the fetch and decode stages of the process. Secondly, the decoded instruction travels through the register file and the ALU or FPU (among others) to comprise a basic datapath. Lastly, caches are also included between the external main memory and the cores. Notice that address generation units and memory management units (MMUs) are also typically incorporated within CPUs to offload tasks regarding the calculation of main memory address or their translation between physical and virtual memory (otherwise, these would have to be calculated for instance by the ALU and *e.g.* require additional clock cycles), but these have been left out for the sake of simplicity.

The different parts that integrate a CPU include:

- **Control unit:** The control unit is in charge of the instruction cycle (also known as fetch-decode-execute cycle). As the name indicates, at every cycle (or more depending on the architecture) an instruction is fetched from the instruction memory (where the program is allocated) and decoded by the instruction decoder in order to pass it forward to the entities that will perform the actual execution. Notice that a branch control mechanism is also included to enable code jumps (such as in loops) along with typically a stack to further enable function calls and returns.
- **Arithmetic logic unit:** The ALU is one of the most essential parts in a processor. It performs arithmetic and bitwise operations on integers (as opposed to the Floating Point Unit, FPU). The ALU requires two inputs: (1) operands and (2) a code representing the operation to be performed. Once the operation is calculated, the result is returned through the output.
- **Registers:** A register file is an array of processor registers that can hold a small amount of data but service the functional components in the CPU extremely fast. Registers are used as a storage for the operands that require manipulation by the ALU, and thus keeping values that are used frequently is a common strategy that compilers apply. In fact, in the C language there is the “register” keyword to hint the compiler that a variable should be put in the register file (however its use is discouraged).

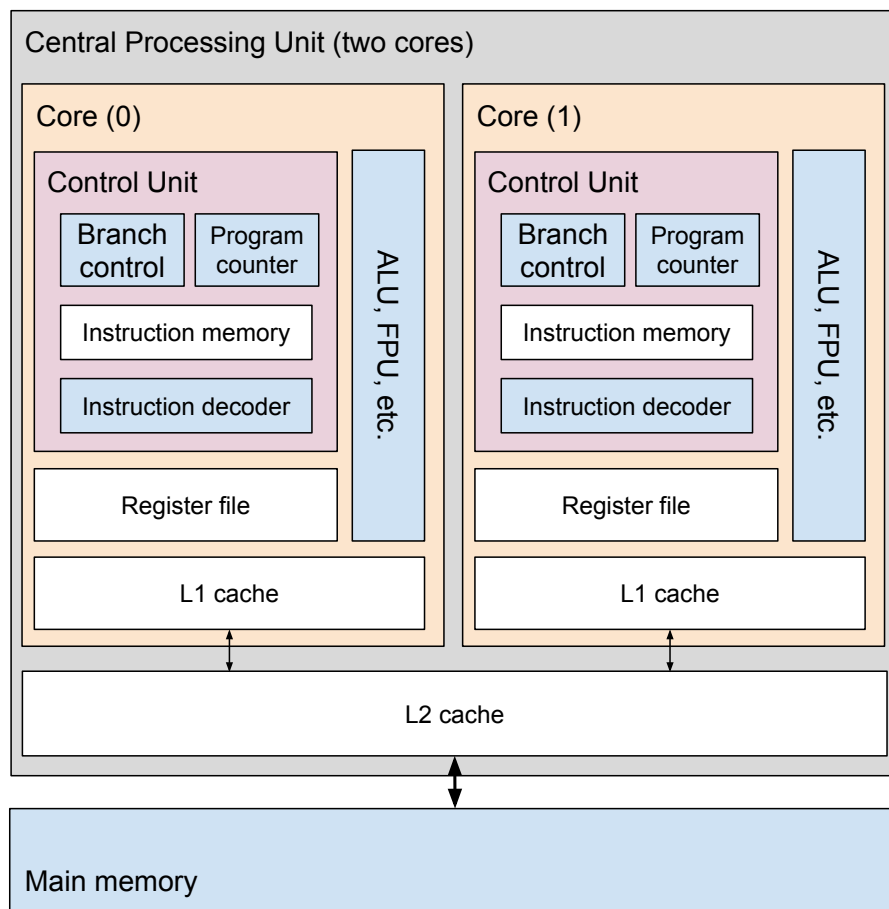


Fig. A.1 Diagram representation of a common CPU. The CPU includes two cores, each of which has its own control unit and datapath, and these communicate with the external main memory throughout the system bus and caches.

- **Clock rate:** The processor, as it is the case in other digital devices, functions at a given frequency. The higher the frequency, the higher the number of cycles that can be processed per seconds, and thus the performance. However, modern architectures have ceased to increase clock rates to gain performance as there are several limiting factors to it, such as (1) pipelined architectures divide instructions into steps (each requiring a fraction of time) and thus clock rates are limited by the longest substep, such that even if the clock ran higher, the performance gain would be none since the other steps would have to wait for the most-time consuming one; (2) designing physically smaller components enables to run higher clocks, however the current scale is hitting physical limitations which makes it nearly impossible to downsize much more; (3) heat dissipation becomes a huge concern as more power is required to increase voltages in order to make circuit switches flip faster.
- **Caches:** In order to reduce the average time of servicing external memory requests of either instructions or data (or both if they are unified), several caches are employed throughout a CPU. Three levels are distinguished and used, with L1 and L2 being the most common. An L1 cache is mostly allocated on a per-core basis, an L2 cache is typically shared among few cores and sometimes another L3 cache is shared for all cores. L1 caches are allocated as close to the register file as possible, trading low latency for small storage size. On one hand, caches closer to the CPU (L1) are usually direct-mapped or have a small degree of associativity in order to increase speed over hit rate. On the other hand, larger caches (L2, L3, *etc.*) have a higher n -way associativity, thus reducing search speed but increasing hit rate.
- **External memory:** Since register files have to be small in order to be close to the functional units in the CPU, an external memory is required to hold both data and code. This is achieved through Random Access Memory (main memory) and secondary storage (for example hard disk drives or solid-state drives). It is noteworthy to mention that the memory hierarchy in a modern computer has become quite complicated, ranging from the register file, through several caches and up until secondary storage. The reasons behind this methodology come clear as soon as the latency is factored in for each of these memory types: while executing an instruction (with the corresponding fetch from register file, *etc.*) can take around one nanosecond, a request from main memory can take up to 10^2 nanoseconds and a request from a hard disk drive can take on the order of 10^6 nanoseconds [141]. The use of intermediate caches can alleviate requests to main memory for a factor of up to 20x.

A.2 Graphic Process Unit (GPU)

The birth of Graphic Processing Units can be traced back to as early as the 1970s [142]. Initially aimed at accelerating the computation of graphics, their design featured a highly parallel structure based on the “Single Instruction, Multiple Data” (SIMD) computer architecture paradigm which enables to process large blocks of data simultaneously. It comes as no surprise that this made them particularly adequate for computing images, as each pixel can be considered a different datapoint where the same series of operations must be computed. Notice that the following literature review will focus mostly on NVIDIA hardware since it has been established as the standard in HPC and scientific computing, with up to a 77% of the market share for desktop GPUs [143], and probably with a higher percentage in the case of research. In fact, at the time of writing, a simple query using the Google Scholar engine shows that searching for “gpu nvidia” returns 150,000 articles, whereas for “gpu amd” the number of articles decreases to roughly 40,000.

In essence, the modern architecture of a GPU is composed of multiple processors (called Streaming Multiprocessors in NVIDIA architecture), each comprising several cores which can access different types of memory interfaces, each being suitable for a particular purpose (*e.g.* device-wide accessible memory is large but slow). Figure A.2 shows the logical and memory model organization of modern GPUs.

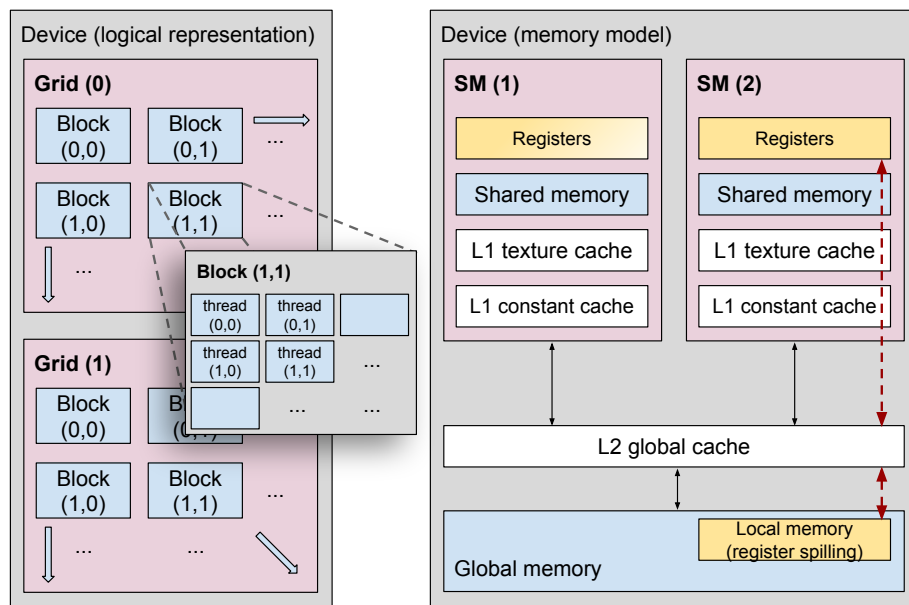


Fig. A.2 (Left) Logical model for CUDA devices. (Right) Memory model. Each block in a grid is composed of several threads, each of which can use the different types of memory available throughout the device.

Notice that Figure A.2 shows both the logical and the memory hierarchy. From the logical perspective, a kernel (*i.e.* a set of instructions) is executed within a grid in a given device. A grid comprises thread blocks which in turn are composed ultimately by single threads. The threads are the minimal working unit who run the code of a kernel. In the other case, the memory hierarchy features several types of memory interfaces, namely: (1) global memory, (2) shared memory, (3) local memory, (4) constant memory and (5) texture memory. The number of different memory types come to show that the achieved performance in GPU computing is strongly related with correct memory utilization. In this line, the optimization of a GPU algorithm is based on maximizing the compute and memory utilization, *i.e.* balancing the number of computational operations (arithmetic, floating point, control flow, *etc.*) with the memory access patterns and the bandwidth. Therefore, the objective is to maximize the number of operations per unit of time as well as the memory throttle.

Moreover, there are also other factors that can be limiting to performance in GPUs just as with any other hardware accelerator: if not enough code is parallelized, if data transfers between host and device are not well orchestrated, if memory accesses are redundant, if the job granularity is not homogeneous, *etc.* Thus, taking advantage of a hardware accelerator such as GPU requires to think in terms of both the algorithmic and HPC perspective, with the latter including not only the specifics to the accelerator device but also the broader scenario.

A.2.1 Compute model

The compute model behind GPUs is aimed at being scalable and able to leverage parallelism as transparently as possible. This is especially true when considering that GPUs can have huge differences in terms of cores and memories, with *e.g.* the most powerful card at the time (namely the NVIDIA RTX 3090) outperforming entry series cards (such as the NVIDIA RTX 2060) by nearly three times more transistors (from 10,800 million to 28,000 million), four times more memory (from 6 to 24 GB) and more than two and half times the number of SMs (from 30 to 82).

In this line, it becomes extremely important to organize computation as seamlessly as possible. This is achieved in CUDA throughout the following logical division of work: a job (kernel) is executed on a GPU in a given grid, divided into batches (blocks) of t threads, where each block of threads is furtherly executed in warp-sized steps inside a SM at a time (with warp size being typically 32 in the latest architectures).

The most appropriate execution path is that where all threads do the same work but to different data (no divergence) and the data is close so it can be fetched in one transaction and distributed to all threads (coalescence, more on that in the next section). Small divergence is tolerable as predication techniques can be applied, however largely different execution paths

will eventually end up in serialization and thus reducing parallel performance massively. Even if the execution paths are homogeneous, performance can still be affected by instruction dependency, which occurs when a requested value in the processing pipeline is not ready and thus can not be processed. This takes place mostly when arithmetic (as well as control ones) operations depend on previous operations that have just been issued and thus the result is not ready. While this should be avoided algorithmically as much as possible, sometimes it is not possible due to the own nature of the algorithm.

Still, other mechanisms exist to “hide” the latency caused by performance drawbacks such as dependent instructions. In particular, each SM can have several blocks active at once (depending on the maximum number of warps that can be active, the amount of registers they require, etc.), and the more there are, the easier it will be for the scheduler to find an eligible warp from a block that is ready to run while others are blocked due to dependencies. This metric is known as occupancy and it measures the number of possible warps that are running divided by the maximum number of warps that can run theoretically. A higher occupancy effectively hides latency up to a certain point, since forcing a higher occupancy than necessary can result in less performance because of each warp having less resources (which eventually can lead to register spilling, *i.e.* the use of global memory for storing local scope variables, or other problems).

A.2.2 Memory model

The memory hierarchy of modern GPUs includes several types of memory blocks and interfaces, each serving a different purpose. Additionally, some of these are cached in different ways to provide faster access (*e.g.* texture memory offers 2D locality caching). Carefully choosing which type is used is key to performance. In particular:

- Global memory (off-chip): This is the largest memory space in the device. It is accessible to all threads (even from different thread blocks and grids). Its main purpose is to allocate large pieces of data that should be read in contiguous and aligned chunks, since if these two properties are satisfied, then multiple threads can be served in a single or fewer grouping transaction(s). This mechanism is called coalescence and is critical to achieving performance. If the memory access patterns (originated by the request of threads) are random or non contiguous, then the device will have no choice but to generate multiple transactions which cause a drop in performance proportional to the number of transactions. Notice that using global memory for the purpose of thread block synchronization is extremely slow and other mechanisms are devised for that matter.

- Texture memory (off-chip global memory cached on-chip): The texture memory is in fact global memory that has been cached for optimizing 2D spatial locality in memory access patterns. This is appropriate specially for graphics applications where it can be assumed that nearby threads will access adjacent parts of the memory, just as it occurs in the mapping of textures to 3D models. Another advantage (among others) is that texture memory offers linear interpolation of adjacent points, which once again is useful for graphics applications. However, it also includes some drawbacks, such as its read-only restriction in order to maintain consistency across the cache (although it can be updated).
- Shared memory (on-chip): Shared memory is allocated per thread block (residing in each SM) and thus can not be accessed device-wide. It offers low latency (nearly 30x less than uncached global memory) at the expense of small size (in the 100 KB range) as it is on-chip. These properties make shared memory very suitable for thread synchronization, user-defined caches, cooperative parallel algorithms (such as reductions), *etc.* Regarding its structure, shared memory is organized in a warp-sized bank grid such that bank 0 contains items 0, $warp_{size}$, $2 * warp_{size}$, $3 * warp_{size}$, $\dots$, in each row. In order to achieve high performance, it is therefore required that memory requests fall either across all banks (*i.e.* one thread one bank, called parallel access) or all threads access the same bank (called broadcast). If two or more threads access a different item on the same bank then it requires serialization, which implies more transactions need to be spawned to service the requests.
- Local memory (off-chip global memory that can be cached): First of all, the definition of local memory is somewhat diffuse, since typically it includes not only strictly-speaking local memory, but also any memory that is available only to threads. That is, each thread includes on-chip memory available only to a single thread where variables are stored (register file). That said, local memory most commonly refers to arrays that are declared per thread or when the number of variables required exceeds that of available registers (known as register spilling). In both cases, “local” memory becomes part of the global memory, thus impacting performance negatively since more transactions are required just to handle stack variables. The biggest difference of local memory managed in global memory is that the memory addresses are resolved by the compiler (as opposed to native global memory). All in all, it becomes obvious that local memory (without considering registers that fit in on-chip memory) should always be avoided.

- Constant memory (cached on-chip): While this memory also resides in the global memory space, it has a small-sized dedicated cache whose main purpose is to store read-only and highly-accessed items (such as constants). This cache is specifically designed to provide faster access if all requests are made to the same location (as in the broadcast of shared memory). Similarly, access to other items gets serialized. This makes the constant memory very useful for storing constants that are requested at the same time by thread blocks.

To complete the memory hierarchy picture, two types of caches can be considered depending on their application scope. These are (1) allocated per SMs or (2) shared by all SMs:

Each SM contains:

- A read-only cache for the constant memory space.
- An L1 cache used to cache reads from global and texture memory.

All SMs share:

- An L2 cache that is used to cache accesses to either local or global memory (including register spills).

In older cards (such as Kepler) the L1 cache shared the same on-chip storage as the shared memory. From Maxwell on, these two were separated, having each their own on-chip space. Lastly, the behaviour of the L1/L2 cache can be partially modified to model which accesses should be cached on load and which not.

A.2.3 Evolution of GPUs

Despite their manifest processing power, the architecture of GPUs keeps evolving. In the last few years, NVIDIA has released several hardware features with each increasing version of the compute capability. Among them, the introduction of warp shuffle functions [144] (starting from KEPLER microarchitectures) enabled huge performance improvements since intra-block communication no longer required any type of barriers nor the use of shared memory (at the expense of more complicated synchronization mechanisms). Many fields benefited from these newly incorporated mechanisms such as string comparison [145], optimal sequence alignment [146] or ciphering [147]. In this line, the proposed research in this thesis also benefits of this type of instruction several times.

A more recent feature released for the latest GPU cards, namely Turing and Ampere includes a new type of core called “tensor core” which enables to perform matrix-multiply-and-accumulate per clock cycle. This is an interesting development not only for machine learning pipelines but also for generic scientific computing, as it opens new paths for speeding up applications where it is possible to find a transformation of the computation flow into matrix multiplications. Recent works including tensor cores (besides machine learning) are so far mostly limited to optimizing core computational processes such as the Fourier transform [148] or specific scan and reduce operations [149], but it is expected that new findings for such features will be available in the near future.

Moore’s law [150], which has been around since 1965, effectively predicted that the number of integrated transistors in a CPU doubled each two years (and thus its performance as well). Even though now Moore’s law is expected to fail due to physical limitations in the manufacturing size of transistors, it has not been as appropriate for GPUs as it has been for CPUs. This is explained through the recently coined Huang’s law [151] which states that the performance of GPUs increases more than two fold per two years due to the newly found applications for GPU computing, which has moved from only aiding in the computation of graphics to an immense number of scientific fields. In any case, the future for High Performance Computing specifically employing GPUs holds promising discoveries.

A.3 Field Programmable Gate Arrays

FPGAs evolved from programmable logic devices (PLD) and programmable read only memories (PROMs). Prior to FPGAs, PLDs and PROMs had to be hard-wired by the manufacturers, which made them non-reprogrammable and therefore less versatile. The key idea behind these types of devices was to offer a board that users could program (and reprogram) on their own without requiring expensive equipment. In 1984, Xilinx introduced the first FPGA [152], which laid out the foundations for the architecture of modern FPGAs: an array of logic blocks composed of LUTs which could perform arbitrary boolean functions. However, at the time it featured only 64 logic blocks and 58 inputs and outputs. Nowadays (and nearly 40 years later), modern FPGAs contain several thousands logic cells, independent blocks of memory and additional fixed-in-silicon circuits to assist the custom logic in more common processing tasks.

FPGAs are similar in nature to ASICs in the sense that both are used to implement custom logic; however, while FPGAs are reprogrammable, ASICs are not (and in fact they are designed for specific purposes). This fundamental difference comes at the cost of performance, *i.e.* the versatility achieved with FPGAs (ease of prototyping, much lower setup

costs, reconfigurability, *etc.*) is traded for smaller performance when compared to ASICs. Several studies have been conducted on the differences in logic density, circuit speed, power consumption, *etc.* For instance, in [153] it is found that the average ratio of required silicon area is 35 times higher in FPGAs (when using pure, custom logic) compared to that of ASICs. However, as described earlier, due to the continuous evolution of FPGAs, the incorporation of hard blocks (fixed-in-silicon circuits) enables to reduce the ratio nearly by half. The study also reflected on power consumption, showing that FPGA power consumption was around 14 times higher on average. Regarding processing speed, this depends mostly on the achieved frequency of the circuit which in turn, depends on the critical path delay (*i.e.* since all components need to work at the same frequency, the board must slow down to the period of the longest path). Notice that parallelization is not considered throughout these studies for it depends on the physical size of the program and this is variable in FPGAs and fixed in ASICs.

Nevertheless, as it is the case with CPUs and GPUs, no hardware device is better than other. Some are more suited for one particular objective than others and vice versa.

A.3.1 Architecture overview

In this section we will describe the architectural design of common FPGAs. Notice that FPGA boards can be highly variable in design depending on their functional goal: high performance, data-intensive applications, power consumption, AI inference, *etc.* However, their architecture is still founded on the same key components: an array of logic blocks, I/O pads and routing channels. These comprise the “soft” blocks (due to their configurability), but additionally, FPGAs may incorporate “hard” blocks which are not as configurable (or have a predefined function), such as embedded processors, memories, PCI-Express controllers, *etc.*

Figure A.3 shows the different components that integrate an FPGA. Notice that only one logic cell is shown in the zoomed part of the logic block (for the sake of simplicity). Normally these include an array of several. In further detail, each component is:

- Logic blocks are the soft components that enable the design of custom logic. Each logic block comprises a number of logical cells (known as slices), and each one is typically composed of an input lookup table (LUT), a multiplexer and a flip flop. In most cases, larger-size LUTs are constructed using smaller LUTs and multiplexers. Since LUTs are so fundamental to FPGAs, their size has been subject of research many times in terms of what the appropriate number of inputs is in relation to efficiency and area occupation results. For instance, in [154] it was found that clustering small LUTs (of size two to three inputs) make better use of the board area, but at the expense of

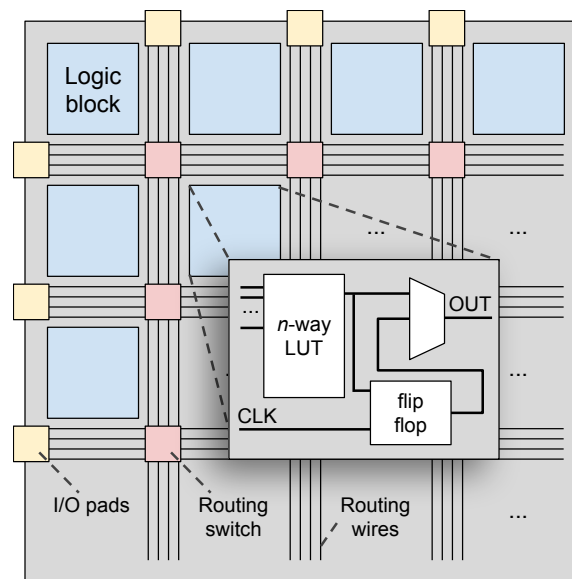


Fig. A.3 Diagram of the architecture of an FPGA. Logic blocks are scattered throughout the board connected by routing wires and switches. Their connection to the outside peripherals is achieved through I/O pads. A zoomed-in logic block shows a single logic cell (simplified).

less performance when compared with larger LUTs (which are more expensive and require more physical area).

- Input and output pads connect the FPGA board with the external peripherals. These are accessed through Input/Output blocks (IOB), which must be placed close to the I/O pads in order to reduce propagation delays [155]. In fact, even if logic blocks are properly placed, an inadequate positioning of IOBs is likely to result in inefficient routing [156].
- Routing elements are the fundamental mechanism to communicate slices in an FPGA. They are composed of routing channels (which can be either horizontal or vertical to navigate the board) and routing switches. The channels provide the connection network, whereas the switches are in charge of directing each path of the design. Besides general-purpose routing infrastructure, additional routing elements exist for delivering clock and reset signals throughout the board while minimising clock skew, *i.e.* the arrival of clock signals to components at different times.
- Hard blocks are embedded circuits (fixed in the board) that serve a particular purpose. Typically hard blocks are designed to perform common functions that if otherwise designed with logic blocks would consume much more area. Some examples include multipliers, adders, embedded memories or even custom processors. Despite their

increased efficiency at their specific design task [157], these blocks can end up wasting a large amount of resources if not utilized.

Appendix B

Sequential computation of k -mers

While the naive k -mer computing algorithm requires up to $3 * k$ operations, it is possible to reduce the complexity of the algorithm by means of algebraic manipulation. In the case of overlapping k -mers, the next hash h_1 shares $k - 1$ letters with the k -mer of the previous hash h_0 . This enables to compute the second hash h_1 in terms of h_0 with few arithmetic operations, as shown below:

$$h_0 = 4^{n-1} * a_0 + 4^{n-2} * a_1 + 4^{n-3} * a_2 + \dots + 4^1 * a_{n-2} + 4^0 * a_{n-1} \quad (\text{B.1})$$

$$h_1 = 4^{n-1} * a_1 + 4^{n-2} * a_2 + 4^{n-3} * a_3 + \dots + 4^1 * a_{n-1} + 4^0 * a_n \quad (\text{B.2})$$

Where a_i is the integer value of nucleotide i in the sequence. Factoring out one power of four:

$$h_1 = 4 * (4^{n-2} * a_1 + 4^{n-3} * a_2 + 4^{n-4} * a_3 + \dots + 4^0 * a_{n-1} + 4^{-1} * a_n) \quad (\text{B.3})$$

The first $n - 1$ terms in the parenthesis are $(h_0 - 4^{n-1} * a_0)$, therefore:

$$h_1 = 4 * ((h_0 - 4^{n-1} * a_0) + 4^{-1} * a_n) \quad (\text{B.4})$$

$$h_1 = 4 * h_0 - 4^n * a_0 + a_n \quad (\text{B.5})$$

The next overlapping hash is always four times the previous one minus $4n$ multiplied by the removed nucleotide (first one in previous k -mer) plus the new nucleotide included in the next k -mer. It can be seen that this algorithmic improvement reduces the number of

arithmetic operations needed per k -mer significantly. Moreover, notice that a simple bitwise implementation is feasible:

$$h_1 = (h_0 \ll 2) + a_n \quad (\text{B.6})$$

A bit shift of two will overflow the two most significant bits which effectively translates into removing $4^n * a_0$ from the previous hash while multiplying by four. Lastly, just the new nucleotide needs to be added. This algorithm is purely linear since it only processes each nucleotide once.

Appendix C

Custom memory pool allocator in ModernGPU merge sort

The sorting time of ModernGPU (merge sort) is faster than that of CUB (radix sort), however the increase in system time due to consecutive memory allocation calls drains the performance (up to 1.5 times more system time in ModernGPU). This is due to the implementation of ModernGPU using several `cudaMalloc` and `cudaFree` calls per sorting run. An `nvprof` execution trace comparing both implementations shows that for the same execution ModernGPU performs 4,052 `cudaMalloc` compared to the 265 `cudaMalloc` from CUB. To overcome this, the memory allocator manager of ModernGPU was modified to take advantage of the fast sorting method while reducing system overhead. Such implementation is available in the github of the CUDAGECKO repository. The same comparison but featuring CUB and the custom modification to the memory allocator manager in ModernGPU can be seen in Table C.1 and C.2.

Table C.1 Comparison of the runtime of the sorting of seeds between Radix sort (CUB library) and mergesort (ModernGPU). Columns “T merge” and “T radix” contain the real and system time (in seconds) for the merge sort and radix sort methods respectively. (*) Value not averaged.

Comparison	MGPU merge.	T merge *	CUB Radix sort.	T radix *
ECOLI B, K12	0.0193431	4.7, 3.8	0.0173032	4.2, 3.7
GALGA18, MELGA20	0.0061253	4.4, 3.8	0.0035522	4.3, 3.6
HOMSAX, MUSMUX	0.0520839	68.0, 10.9	0.0561467	68.2, 8.4
HOMSA1, GORGO1	0.0883965	211.0, 22.3	0.0966461	206.4, 15.0

Table C.2 Comparison of the runtime of the sorting of seeds between Radix sort (CUB library) and mergesort (ModernGPU) with the custom memory allocator implementation. Columns “T merge” and “T radix” contain the real and system time (in seconds) for the merge sort and radix sort methods respectively. (*) Value not averaged.

Comparison	MGPU custom.	T merge *	CUB Radix sort.	T radix *
ECOLI B, K12	0.013669	4.2, 3.7	0.0173032	4.2, 3.7
GALGA18, MELGA20	0.002285	4.2, 3.6	0.0035522	4.3, 3.6
HOMSAX, MUSMUX	0.043812	62.2, 7.5	0.0561467	68.2, 8.4
HOMSA1, GORGO1	0.077244	194.4, 13.7	0.0966461	206.4, 15.0

the runtime is reduced to nearly 92% of the original, which translates into approximately 20 seconds in the large-sized comparisons on a 980 GTX graphics card. The code for the memory allocator is shown in Code snippet 1.

Listing C.1 File context.hxx of the ModernGPU implementation.

```
virtual void* alloc(size_t size, memory_space_t space) {
    void* p = nullptr;
    if (size) {
        cudaError_t result;
        if (memory_space_device == space)
        {
            if (mptr != NULL)
            {
                uint64_t new_address = mptr->address + 32 - (mptr->address % 32);
                if (new_address + (uint64_t) size >= mptr->limit) { fprintf(stderr, "Not enough memory in pool\n"); exit(-1); }
                uint64_t t_occupied = (new_address + (uint64_t) size) - mptr->address;
                allocs.push(t_occupied);
                p = (void *) (mptr->mem_ptr + new_address);
                mptr->address += t_occupied;
                result = cudaSuccess;
            }
            else
                result = cudaMalloc(&p, size);
        }
        else
            result = cudaMallocHost(&p, size);

        if (cudaSuccess != result) throw cuda_exception_t(result);
    }
    return p;
}
```

Listing C.2 Freeing procedure for the custom memory allocator.

```
virtual void free(void* p, memory_space_t space) {
    if (p) {
        cudaError_t result;
        if (memory_space_device == space)
        {
            if (mptr != NULL)
            {
                uint64_t last_size = allocs.top();
                allocs.pop();
                mptr->address -= last_size;
                result = cudaSuccess;
            }
            else
                result = cudaFree(p);
        }
        else
    }
```

```
        result = cudaFreeHost(p);  
        if(cudaSuccess != result) throw cuda_exception_t(result);  
    }  
}
```

Listing C.3 Memory pool structure.

```
#include <inttypes.h>  
  
typedef struct mem_pool{  
  
    char * mem_ptr;  
    uint64_t address;  
    uint64_t limit;  
  
} Mem_pool;
```


Appendix D

List of publications supporting this thesis.

The list of publications that support this thesis are shown next including title, citation and abstract.

D.1 Publication 1

Pérez-Wohlfeil, Esteban, Sergio Diaz-del-Pino, and Oswaldo Trelles. "Ultra-fast genome comparison for large-scale genomic experiments". Sci Rep 9, 10274 (2019). <https://doi.org/10.1038/s41598-019-46773-w>

D.1.1 Abstract

In the last decade, a technological shift in the bioinformatics field has occurred: larger genomes can now be sequenced quickly and cost effectively, resulting in the computational need to efficiently compare large and abundant sequences. Furthermore, detecting conserved similarities across large collections of genomes remains a problem. The size of chromosomes, along with the substantial amount of noise and number of repeats found in DNA sequences (particularly in mammals and plants), leads to a scenario where executing and waiting for complete outputs is both time and resource consuming. Filtering steps, manual examination and annotation, very long execution times and a high demand for computational resources represent a few of the many difficulties faced in large genome comparisons. In this work, we provide a method designed for comparisons of considerable amounts of very long sequences that employs a heuristic algorithm capable of separating noise and repeats from conserved fragments in pairwise genomic comparisons. We provide software implementation that computes in linear time using one core as a minimum and a small, constant memory footprint. The method produces both a previsualization of the comparison and a collection of indices to

drastically reduce computational complexity when performing exhaustive comparisons. Last, the method scores the comparison to automate classification of sequences and produces a list of detected syntenic blocks to enable new evolutionary studies.

D.2 Publication 2

Pérez-Wohlfeil, Esteban, and Oswaldo Trelles. "Precise and parallel pairwise metagenomic comparisons." *Journal of Computational Biology* 25, no. 8 (2018): 841-849. <http://doi.org/10.1089/cmb.2018.0081>

D.2.1 Abstract

The comparison and assessment of similarity across metagenomes are still an open problem. Uncultivated samples suffer from high variability, thus making it difficult for heuristic sequence comparison methods to find precise matches in reference databases. Finer methods are required to provide higher accuracy and certainty, although these come at the expense of larger computation times. Therefore, in this work, we present our software for the highly parallel, fine-grained pairwise alignment of metagenomes. First, an analysis of the computational limitations of performing coarse-grained global alignments in parallel manner is described, and a solution is discussed and employed by our proposal. Second, we show that our development is competitive with state-of-the-art software in terms of speed and consumption of resources, while achieving more accurate results. In addition, the parallel scheme adopted is tested, depicting a performance of up to 98% efficiency while using up to 64 cores. Sequential optimizations are also tested and show a speedup of 9x over our previous proposal.

D.3 Publication 3

Pérez-Wohlfeil, Esteban, Oswaldo Trelles and Nicolas Guil. “Irregular alignment of arbitrarily long DNA sequences on GPU”, *The Journal of Supercomputing* (2022): 1-30. <https://doi.org/10.1007/s11227-022-05007-z>

D.3.1 Abstract

The use of Graphics Processing Units to accelerate computational applications is increasingly being adopted due to its affordability, flexibility and performance. However, achieving top performance comes at the price of restricted data-parallelism models. In the case of sequence alignment, most GPU-based approaches focus on accelerating the Smith-Waterman dynamic programming algorithm due to its regularity. Nevertheless, because of its quadratic complexity, it becomes impractical when comparing long sequences, and therefore heuristic methods are required to reduce the search space. We present GPUGECKO, a CUDA implementation for the sequential, seed-and-extend sequence-comparison algorithm, GECKO. Our proposal includes optimized kernels based on collective operations capable of producing arbitrarily long alignments while dealing with heterogeneous and unpredictable load. Contrary to other state-of-the-art methods, GPUGECKO employs a batching mechanism that prevents memory exhaustion by not requiring to fit all alignments at once into the device memory, therefore enabling to run massive comparisons exhaustively with improved sensitivity while also providing up to 6x average speedup w.r.t. the CUDA acceleration of BLASTN.

Appendix E

Resumen en Español - Summary in Spanish

E.1 Introducción

Debido al auge de las tecnologías de adquisición de datos, se necesitan nuevas herramientas informáticas capaces de procesar el gran volumen de información generado. En especial, algunos campos científicos, como por ejemplo la biología computacional o el aprendizaje de máquinas, se han visto completamente desbordados por una marea de datos; hasta el punto en que el incremento en la demanda de procesamiento de dichos datos ha hecho patente la necesidad de diseñar nuevos algoritmos. Este fenómeno no sólo ha afectado a sistemas de computación monoprocesador, sino también a todo tipo de arquitecturas y aceleradores hardware específicos, tales como las FPGA (Field-programmable gate array) o las GPUs (Graphics Processing Units). En particular, éstos últimos han sido objeto de una explosión tanto en adopción como en uso prolongado, especialmente para asistir en la llamada computación de propósito general (GPGPU o General-Purpose computing on GPUs) como reemplazo de costosos sistemas multiprocesador, supercomputadores, etc. Esto ha sido posible gracias a la gran capacidad de paralelismo de estos dispositivos, además de su eficiencia y bajo costo de adquisición y mantenimiento que, comparados con supercomputadores tradicionales o los servicios de renting de Cloud Computing, los hacen aún más atractivos para grupos de investigación tanto grandes como pequeños.

Existen numerosas disciplinas científicas que hacen uso del GPGPU, como por ejemplo simulación de fluidos, procesamiento de vídeo, aprendizaje computacional, etc. Sin embargo, algunas disciplinas aún están empezando a hacer uso de dicha metodología. Tal es el caso

de la genómica comparativa, y en particular el alineamiento de secuencias. A pesar de que dicha disciplina es a la vez una gran productora y consumidora de datos (por ejemplo, sólo el genoma humano tiene una longitud de tres mil millones de nucleótidos o “letras”), la mayoría de los algoritmos empleados suelen diseñarse sólo para su ejecución en máquinas secuenciales y supercomputadores utilizando paralelismo de tarea, y en raras ocasiones se realiza su adaptación y optimización para paralelismos de grano fino y, aún menos, aceleradores gráficos.

El caso de la genómica comparativa y el alineamiento de secuencias de ADN es de especial interés dado que es un problema central para muchas aplicaciones que tienen impacto directo en la salud humana: agricultura modificada genéticamente, diseño de vacunas y respuesta a fármacos, medicina de precisión, etc. Esto implica que su aceleración computacional es de gran importancia tanto para la comunidad científica como para la humanidad en su conjunto. Más aún, el uso de aceleradores gráficos permitiría abaratar costes y facilitar que muchos más científicos y científicas investiguen dichas disciplinas pese a la alta complejidad computacional. Sin embargo, a la par del creciente interés, se está observando un incremento exponencial en el número de secuencias de ADN publicadas en bases de datos de acceso público, tanto en número como en longitud de las mismas secuencias. Esto se traduce en la creación de una serie de cuellos de botella computacionales para los cuales los algoritmos tradicionales no son capaces de dar respuesta ni en tiempo aceptable ni utilizando un consumo de recursos hardware razonable.

En el caso concreto del alineamiento de secuencias por pares, el objetivo es encontrar regiones o zonas estructuralmente similares entre dos cadenas de ADN (que pueden ser extremadamente largas), dado que de ser así existe una mayor probabilidad de que cumplan una función biológica similar y/o estén relacionadas evolutivamente. Dicha tarea puede verse como una variante multidimensional del famoso problema de la búsqueda de la subsecuencia común más larga, donde en vez de encontrar sólo las subsecuencias más largas, se buscan todas aquellas que puedan satisfacer una serie de restricciones (por ejemplo, de cierta longitud y similitud). Es bien conocido que la variante más sencilla, *i.e.*, aplicada a tan sólo dos cadenas, se puede resolver de forma óptima utilizando el método de programación dinámica (debido a su posible descomposición en subestructuras óptimas y superposición de subproblemas) cuya complejidad en tiempo no es inferior a $O(n * m)$, siendo n y m la longitud de las secuencias. Pese a la exactitud de este método, su complejidad cuadrática hace imposible su aplicación práctica en tiempo razonable cuando las secuencias alcanzan cierta longitud, especialmente en el caso de cromosomas de mamíferos, que a su vez es uno

de los más frecuentes.

Más aún, debido a la naturaleza arbitraria de las secuencias de ADN, la optimización y aceleración paralela de la comparación de grandes secuencias plantea interesantes retos computacionales, incluyendo granularidades altamente heterogéneas, cargas impredecibles, necesidad de mecanismos de balanceo y sincronización, limitación del rendimiento o throughput por el ancho de banda de la memoria, etc. Por lo tanto, para lograr un alto rendimiento, no sólo basta con el diseño de algoritmos eficientes, sino que, además, los algoritmos deben adaptarse al modelo de procesamiento de la arquitectura subyacente, lo que puede representar esquemas de paralelismo computacionales completamente diferentes (por ejemplo, paralelismo de tareas y datos) y a menudo requieren incluso el rediseño completo de los propios algoritmos.

Por todo lo descrito previamente, en esta tesis se aborda un recorrido computacional del problema de la comparación de secuencias haciendo uso de optimizaciones algorítmicas en diferentes arquitecturas, con distintos niveles de precisión y granularidad. En particular, se estudia el problema para (1) máquinas de núcleo único, (2) sistemas concurrentes de memoria compartida, (3) aceleradores gráficos y (4) finalmente, de forma más abstracta, se estudia el mismo problema desde la perspectiva de planificadores de tareas en supercomputadores utilizando aprendizaje computacional.

E.2 Primera contribución

La primera contribución presenta un desarrollo formal basado en fundamentos analíticos que permite la comparación de secuencias con un espacio de búsqueda prácticamente ilimitado gracias a su metodología basada en hashing y a su complejidad temporal estrictamente lineal $O(n)$. La motivación de este trabajo nace como respuesta a la necesidad de los investigadores de realizar estudios evolutivos donde se requiere comparar no sólo una sino múltiples secuencias. En particular, el caso más común se centra en la comparación de todos los cromosomas correspondientes a varios genomas, por ejemplo, el de humano con el de gorila, lo que se traduce en $23 * 24 = 552$ comparaciones de cromosomas (puesto que el humano y el gorila tienen 23 y 24 pares de cromosomas, respectivamente). Dado el gran tamaño de estas secuencias y el hecho mencionado previamente de que el alineamiento óptimo de secuencias es fundamentalmente un problema cuadrático, resulta inmediato ver el considerable tiempo de ejecución que puede requerir completar dichas ejecuciones. Más aún, en el momento que se incluyen más especies en la ecuación (por ejemplo, el ratón con 20 cro-

mosomas), el número de comparaciones aumenta a $23 * 24 * 20 = 11040$. Si para un estudio evolutivo se considerasen sólo genomas de mamíferos, el número de especies involucradas alcanzaría alrededor de 5000 a 6000, lo que se traduciría es un espacio de búsqueda intratable.

Por lo tanto, con el objetivo en mente de facilitar los estudios evolutivos a científicos y científicas, la primera contribución presenta un método capaz de detectar las regiones donde se encuentra la mayor cantidad de subsecuencias comunes entre dos secuencias en tiempo estrictamente lineal y utilizando memoria constante que no depende de la entrada. Esto permite utilizar el método propuesto como una fase de cribado en comparaciones de cromosomas múltiples para eliminar aquellas comparaciones que no posean suficientes regiones estructuralmente similares, y tras superar la fase de cribado satisfactoriamente, utilizar métodos de comparación de grano más fino sólo en las regiones detectadas en vez de las secuencias completas. Este proceso permite reducir el espacio de búsqueda de cuadrático a lineal a cambio de una leve pérdida de precisión en el número de regiones o alineamientos similares encontrados. A su vez, la contribución propone una métrica acotada para clasificar de forma automática el resultado de una comparación, de forma que el sistema se pueda utilizar en combinación con otros algoritmos de forma no supervisada. Esto es especialmente importante debido a que la inspección y refinado manual de cientos de miles de comparaciones no es humanamente posible.

De forma breve, el algoritmo procede de la siguiente forma: (1) indexado de una de las secuencias utilizando muestreo a nivel de subsecuencia; (2) emparejado parcialmente inexacto de las subsecuencias (de la otra secuencia) con el índice; (3) filtrado de subsecuencias repetidas acorde al desarrollo formal; (4) agrupamiento o clustering heurístico de las subsecuencias emparejadas únicas; (5) selección de óptimos locales o centroides y finalmente (6) postproceso y puntuación de la comparación junto a la detección y clasificación de eventos evolutivos a gran escala. En conjunto, el algoritmo propuesto es capaz de reducir el tiempo de ejecución en un factor de más de 10x y el consumo de memoria en aproximadamente 50x comparado con el estado del arte. Más aún, cuanto más grandes las secuencias comparadas, mayor es la aceleración obtenida debido a la complejidad estrictamente lineal y el uso de memoria constante.

Finalmente, el método propuesto es candidato para una implementación paralela a nivel de tarea, debido a la granularidad gruesa de los datos. En particular, dado el gran tamaño de los genomas (varios cientos de megabytes) y el corto tiempo de ejecución requerido por el método propuesto (alrededor de un minuto por comparación), se desestima el par-

alelismo de datos en favor de una metodología distribuida donde cada nodo ejecuta múltiples comparaciones independientes. Como trabajo futuro, en esta tesis se propone el estudio de la asignación óptima de dichas comparaciones a un conjunto de máquinas dadas las restricciones de latencia de transferencia y localidad de los datos y el uso superpuesto de índices en la comparación, ya que, al comparar cada secuencia con un número variable de secuencias, los índices pueden no necesitar ser recalculados dependiendo de la latencia de transferencia.

E.3 Segunda contribución

La metagenómica se puede definir como el análisis genético de muestras recogidas directamente de un origen natural y/o ambiental. En esta línea, la segunda contribución describe un enfoque paralelo que utiliza máquinas de memoria compartida para lograr una alta sensibilidad en la comparación de secuencias metagenómicas incompletas y de baja calidad. Al contrario que los genomas tradicionales, que son cultivados de forma individual y aislada en un laboratorio, las muestras metagenómicas proceden de un cultivo natural (por ejemplo, un lago o la saliva de un perro) donde conviven numerosos organismos. Esto implica que una muestra metagenómica puede contener cientos de especies (o más) representadas de forma parcial y no uniforme. Más aún, debido a los numerosos procesos que intervienen en la producción de una muestra metagenómica, entre ellas el propio proceso tecnológico de secuenciación (llamado Next Generation Sequencing, o NGS) o el inevitable deterioro causado por la recolección, depósito y tratamiento, las muestras metagenómicas suelen representar la población de especies original de forma no homogénea, arbitraria e incompleta. A razón de esta pérdida de calidad, el posterior análisis y clasificado computacional puede requerir de un mayor grado de precisión en comparación con la comparativa genómica estándar para evitar la introducción de todavía más ruido.

Los enfoques existentes para la metagenómica comparativa se basan en utilizar bases de datos de genomas para mapear el contenido de las muestras, y luego compararlas entre ellas, o bien comparándolas directamente sin utilizar ningún tipo de intermediario. En el primer caso, se establece una relación de composición de ciertos genomas en las muestras (típicamente junto a valores de frecuencia o representación) y finalmente se comparan dichas presencias entre muestras independientes de metagenomas, lo que puede introducir errores en la comparación, por ejemplo, debido a bases de datos incompletas. En el segundo caso, las muestras metagenómicas son comparadas directamente entre ellas para establecer un grado

de similitud entre ellas sin requerir el conocimiento previo de cuál es su composición. Como resultado de calcular la similitud entre muestras, es posible inferir un orden o agrupamiento entre las mismas (por ejemplo, separar muestras provenientes de animales sanos o enfermos).

Por lo tanto, la segunda contribución presenta un método paralelo capaz de estimar la similitud entre muestras metagenómicas utilizando alineamiento de secuencias óptimo, sin requerir optimización de hiperparámetros y sin introducir sesgo o parcialidad en los resultados. Esto se consigue a través de un proceso iterativo en el que una región candidata a ser alineada de forma óptima pasa por diferentes filtros de precisión, *i.e.* empezando por detección de semillas, seguido de extensión sin gaps (asumiendo sólo operaciones de sustitución en las cadenas de ADN) y finalmente extensión con gaps (incluyendo inserciones y deleciones). Este procedimiento progresivo permite obtener un alto grado de precisión incluso con grandes espacios de búsqueda de varios billones de alineamientos. Esto es posible debido a que cada etapa tiene una complejidad en tiempo y espacio muy inferior a la siguiente, y una región candidata sólo progresa de etapa si cumple una serie de requisitos estadísticos.

Desde la perspectiva del paralelismo, el método emplea (1) un balanceador de carga customizado dependiente de la distribución subyacente de los metagenomas, (2) la construcción en paralelo de un índice de candidatos a alineamiento y (3) una serie de optimizaciones secuenciales que permiten acotar el tiempo y espacio requerido para la computación de alineamientos óptimos. En conjunto, la propuesta permite un paralelismo de grano fino con alta eficiencia en máquinas de memoria compartida y plantea la posibilidad de escalar utilizando paralelismo a nivel de tarea en nodos distribuidos. En términos de aceleración, el método es hasta 3x veces más rápido y escala con mucha mayor eficiencia que otros algoritmos similares del estado del arte, a la vez que puede detectar una mayor cantidad de alineamientos respecto de longitud e identidad.

E.4 Tercera contribución

La tercera contribución nace como contraposición a la primera: si la primera resolvía el problema de la comparación de grandes secuencias de forma heurística y con recursos computacionales limitados, la tercera contribución resuelve el mismo problema pero de forma exhaustiva al explotar el paralelismo masivo de los aceleradores gráficos. En particular, se presenta una implementación completa en GPU utilizando CUDA del algoritmo de alineamiento de secuencias GECKO. Debido a la naturaleza relativamente caótica del ADN, la mayoría de

las propuestas de la literatura sobre uso de GPUs para alineamiento de secuencias se basa en implementaciones eficientes del algoritmo de programación dinámica Needleman-Wunsch o Smith-Waterman, principalmente debido a su alta regularidad y facilidad de acotado. De hecho, en la literatura se reportan aceleraciones de hasta 100x a pesar de las dependencias de datos tradicionales de las matrices de programación dinámica. Por el contrario, los algoritmos basados en extensión de semillas apenas han sido objeto de implementación en aceleradores gráficos debido a su alta irregularidad, particularmente proveniente de cuatro grandes fuentes: (1) enorme longitud de las secuencias, (2) un número desconocido y explosivo de semillas potencialmente cuadrático en términos de la entrada, (3) longitud arbitraria de los alineamientos resultantes y (4) solapamiento de los alineamientos con las propias semillas. Dichas fuentes de irregularidad resultan en problemas de desbordamiento de búfer, balanceo, sincronización y computación redundante, los cuales son extremadamente importantes a la hora de obtener el máximo rendimiento de los dispositivos GPU.

El método propuesto en la tercera contribución es un algoritmo integral en GPU que incluye kernels optimizados para maximizar la eficiencia de los accesos y el ancho de banda de la memoria (coalescencia), la ocupación, etc., a la vez que minimizar la dependencia entre instrucciones y la divergencia entre hilos. Esto se consigue a través de una técnica de procesamiento por lotes (batching) y una serie de kernels basados en operaciones colectivas (reducciones, scan, scatter, etc.). Finalmente, el método se acompaña de una metodología para su uso en el servicio CUDA MPS, permitiendo o bien aumentar la eficiencia a través de la ejecución múltiple de comparaciones en una misma GPU o liberando recursos de la misma para otras aplicaciones.

En conjunto, el método es capaz de (1) tratar con espacios de búsqueda masivos de forma exhaustiva, (2) computar la comparación de secuencias íntegramente en el dispositivo dejando libre la CPU y (3) solventar las fuentes de irregularidad existentes en los algoritmos de extensión de semillas. Esto resulta en una aceleración de entre 6x y 10x en comparación con el estado del arte (dependiendo del tipo de procesado) y holgadamente por encima de un orden de magnitud comparado con el algoritmo secuencial en CPU, a la vez que proporcionando mayor sensibilidad en la detección de alineamientos. Por otro lado, se demuestra cómo el algoritmo tiene una complejidad lineal en el número de semillas detectadas, las cuales correlacionan con la similitud y longitud de las secuencias de entrada. Desde la perspectiva de la biología computacional, el método proporciona a los investigadores con una nueva herramienta capaz de calcular comparaciones de cromosomas de organismos superiores de forma exhaustiva, *i.e.* proporcionando alineamientos, en menos de un minuto de media

utilizando tarjetas gráficas convencionales.

E.5 Cuarta contribución

Por último, se presenta una contribución que combina todos los conocimientos anteriores, resultando en una propuesta sobre planificación de tareas asistida por métodos de aprendizaje computacional para mejorar la asignación de recursos y el rendimiento en supercomputadores dedicados a la comparación de secuencias. En particular, se estudia el problema de estimar la cantidad de recursos computacionales (concretamente, CPUs y memoria) y el tiempo total requerido (makespan) por un proceso de comparación de genomas. Es bien conocido que los usuarios de supercomputadores a menudo sobreestiman la cantidad de recursos que necesitan con objeto de evitar una posible cancelación del trabajo. Más aún, cuando se trata de campos científicos interdisciplinares, como es el caso de la biología computacional, dicho problema se agudiza debido a que los usuarios suelen poseer más competencias en un campo que en otro, haciendo por tanto más difícil la estimación. Por ello, automatizar y mejorar la estimación (incluso levemente) permitiría una mayor eficiencia y aprovechamiento de grandes supercomputadores, reduciendo tiempos de espera y aumentando el rendimiento.

El problema se aborda empezando por estimar adecuadamente el número de palabras comunes entre dos secuencias de ADN, ya que dicho número se correlaciona con el tiempo de ejecución. Esto es debido a que los alineamientos suelen incorporar conjuntos de palabras iguales (semillas), y por lo tanto a mayor número de palabras comunes, mayor cantidad de alineamientos que deben ser computados, y, por ende, tiempo. Para estimar la cantidad de palabras comunes se utilizan las frecuencias de n-gramas de pequeño tamaño ya que estos se pueden computar en tiempo sub-lineal y con un consumo de memoria del orden de kilobytes. La metodología propuesta utiliza modelos tradicionales de aprendizaje computacional (en particular Random Forest y Boosting) y propone un nuevo diseño de red neuronal basado en regresión no lineal. Estos modelos son entrenados con más de 200,000 comparaciones de secuencias de cromosomas de todo tipo de especies y organismos superiores. Más aún, las comparaciones son realizadas utilizando la metodología de comparación de secuencias en GPU presentada en la tercera contribución, la cual permite hacer viable los experimentos al reducir los tiempos de ejecución drásticamente.

Los resultados de la última contribución se validan mediante experimentos basados en (1) precisión y generalización de los modelos dependiendo de la organización taxonómica

de las especies, (2) análisis y explicación del modelo utilizando Contribution Analysis, (3) discusión y análisis del problema en su forma de clasificación o regresión y finalmente (4) simulación sintética y con trazas reales de un supercomputador para estudiar la mejora en el rendimiento dadas las predicciones del modelo. El resultado principal es que el modelo de red neuronal propuesto alcanza un 94% del rendimiento óptimo (*i.e.*, conociendo la asignación de recursos ideal), lo que reduce en aproximadamente un 15% el makespan total de la traza completa.

E.6 Conclusiones y trabajo futuro

En esta tesis se han presentado cuatro trabajos relacionados con los campos de la Arquitectura de Computadores, la Informática y la Biología Computacional. En concreto, se describe el uso de diferentes arquitecturas de hardware, incluyendo monoprocesadores, máquinas de memoria compartida y Unidades de Procesamiento Gráfico para acelerar la tarea computacionalmente costosa del alineamiento de secuencias, que es fundamental en los campos de las Ciencias de la Vida, la Bioinformática y la Medicina de Precisión, entre otros. También se incluye un trabajo adicional con un nivel de abstracción superior que aúna las tres disciplinas, particularmente el estudio de la mejora de la eficiencia de los planificadores asistidos por modelos de aprendizaje automático para sistemas de comparación de secuencias a gran escala.

En resumen, los resultados de esta tesis son dobles: desde una perspectiva computacional, se proponen nuevas metodologías de High Performance Computing, estructuras de datos y esquemas de paralelismo para una variedad de arquitecturas hardware; y desde la perspectiva de la genómica comparativa, se ha logrado reducir la complejidad computacional del problema de comparación de secuencias tanto para métodos exhaustivos como heurísticos, ofreciendo distintos niveles de granularidad y precisión dependiendo de la arquitectura empleada. De forma general, también se ha proporcionado a los investigadores un nuevo conjunto de herramientas capaces de manejar espacios de búsqueda anteriormente impensables, lo que se traduce en la apertura de nuevas vías de investigación y resultados.

La primera contribución se acompaña de la siguiente publicación:

- **Pérez-Wohlfeil, Esteban**, Sergio Diaz-del-Pino, and Oswaldo Trelles. "Ultra-fast genome comparison for large-scale genomic experiments." *Scientific reports* 9, no. 1 (2019): 1-10. (Q1, IF 3.99).

La segunda contribución se acompaña de las siguientes publicaciones:

- **Pérez-Wohlfeil, Esteban,** and Oswaldo Trelles. "Precise and parallel pairwise metagenomic comparisons." *Journal of Computational Biology* 25, no. 8 (2018): 841-849 (Q4, IF 0.89).
- **Pérez-Wohlfeil, Esteban,** Oscar Torreno, and Oswaldo Trelles. "Pairwise and incremental multi-stage alignment of metagenomes: a new proposal." In *International Conference on Bioinformatics and Biomedical Engineering*, pp. 74-80. Springer, Cham, 2017 (International conference).
- **Pérez-Wohlfeil, Esteban,** Jose A. Arjona-Medina, Oscar Torreno, Eugenia Ulzurrun, and Oswaldo Trelles. "Computational workflow for the fine-grained analysis of metagenomic samples." *BMC genomics* 17, no. 8 (2016): 351-361 (Q1, IF 3.72).

La tercera contribución se acompaña de la siguiente publicación:

- **Pérez-Wohlfeil, Esteban,** Oswaldo Trelles and Nicolas Guil. "Irregular alignment of arbitrarily long DNA sequences on GPU", *The Journal of Supercomputing* (2022): 1-30. (Q2, IF 2.42).

La cuarta contribución fue presentada en un congreso a nivel nacional y se postula como trabajo futuro su continuación:

- **Pérez-Wohlfeil, Esteban** and Oswaldo Trelles. "Towards the efficient scheduling of sequence comparisons using Machine Learning", *Jornadas Sarteco 2021, VI Congreso Nacional de Informática (CEDI)*, Málaga, Spain, Sep 2021 (Conferencia nacional).

Respecto del trabajo futuro, nuevas líneas de investigación quedan abiertas debido a los resultados obtenidos en esta tesis. Es el deseo del autor continuar con estos trabajos siempre y cuando sea posible. En particular:

- Inclusión de un kernel cooperativo de alineación con gaps en la contribución basada en GPU. Esto permitiría mejorar la precisión en la detección de alineamientos, particularmente en secuencias muy remotas.
- Continuación del estudio sobre el espacio paramétrico y su optimalidad para comparaciones de metagenomas dependiendo de la parcialidad e incompletitud de los datos, lo cual podría llevar a más reducciones en el coste computacional de los alineamientos.

- Inclusión de más genomas en la fase de entrenamiento y validación del modelo de aprendizaje computacional utilizado para asistir en la planificación de tareas en supercomputadores. Esto permitiría mejorar la predicción y reducir la varianza debido a la proporción de especies por taxonomía.
- Extensión de la simulación utilizada para la planificación de tareas, entre otros utilizando diferentes colas de planificación, técnicas de backfill, más trazas de supercomputadores, *etc.*
- Diseño de una interfaz de paso de mensajes para optimizar la computación de múltiples comparaciones de secuencias descrita en la primera contribución en nodos distribuidos.
- Refinado de la distancia propuesta en la primera contribución para la clasificación automática de comparaciones de secuencias, particularmente incorporando número y tamaño de los eventos evolutivos.