DOCTORAL THESIS

High-Order Epistasis Detection in High Performance Computing Systems

Christian Ponte Fernández

2022

UNIVERSIDADE DA CORUÑA

High-Order Epistasis Detection in High Performance Computing Systems

---

Christian Ponte Fernández

DOCTORAL THESIS

22 April 2022

PhD Advisors:

María José Martín Santamaría

Jorge González Domínguez

PhD Program in Information Technology Research

UNIVERSIDADE DA CORUÑA

Dra. María José Martín Santamaría
Catedrática de Universidad
Dpto. de Ingeniería de Computadores
Universidade da Coruña

Dr. Jorge González Domínguez
Profesor Titular de Universidad
Dpto. de Ingeniería de Computadores
Universidade da Coruña

CERTIFICAN

Que la memoria titulada "High-Order Epistasis Detection in High Performance Computing Systems" ha sido realizada por D. Christian Ponte Fernández bajo nuestra dirección en el Departamento de Ingeniería de Computadores de la Universidade da Coruña, y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Investigación en Tecnologías de la Información.

En A Coruña, a 22 de abril de 2022

Fdo.: María José Martín Santamaría
Directora de la Tesis Doctoral

Fdo.: Jorge González Domínguez
Director de la Tesis Doctoral

Fdo.: Christian Ponte Fernández
Autor de la Tesis Doctoral

A todos los que de una forma u otra
contribuisteis a esta tesis

Acknowledgements

First, I would like to thank my advisors, María and Jorge, who have been patient enough and supported me throughout this thesis. I am lucky that they chose me as their pre-doctoral fellow. This sentiment can also be extended to Antonio, who helped us during the early stages of this thesis with the experiment design and data simulation of the review study. I would also like to thank my family for the support that they showed throughout my academic career. Last but not least, I would like to extend the acknowledgement and gratitude to the whole Computer Architecture Group for all the good moments that I had working there.

I also want to acknowledge the funders of this work: the Ministry of Economy and Competitiveness of Spain (ref. TIN2016-75845-P), the Ministry of Science and Innovation of Spain (ref. PID2019-104184RB-I00 / AEI / 10.13039/501100011033) and the Xunta de Galicia (refs. ED431G/01, ED431C 2017/04, ED431G 2019/01 and ED431C 2021/30).

Christian Ponte Fernández

Resumo

Nos últimos anos, os estudos de asociación do xenoma completo (Genome-Wide Association Studies, GWAS) están a gañar moita popularidade de cara a buscar unha explicación xenética á presenza ou ausencia de certas enfermidades nos humanos. Hai un consenso nestes estudos sobre a existencia de interaccións xenéticas que condicionan a expresión de enfermidades complexas, un fenómeno coñecido como epistasia. Esta tese céntrase no estudo deste fenómeno empregando a computación de altas prestacións (*High-Performance Computing*, HPC) e dende a súa perspectiva estadística: a desviación da expresión dun fenotipo como a suma dos efectos individuais de múltiples variantes xenéticas. Con este obxectivo desenvolvemos unha primeira ferramenta, chamada *MPI3SNP*, que identifica interaccións de tres variantes a partir dun conxunto de datos de entrada. *MPI3SNP* implementa unha busca exhaustiva empregando un test de asociación baseado na Información Mutua, e explota os recursos de clústeres de CPUs ou GPUs para acelerar a busca. Coa axuda desta ferramenta avaliamos o estado da arte da detección de epistasia a través dun estudo que compara o rendemento de vintesete ferramentas. A conclusión máis importante desta comparativa é a incapacidade dos métodos non exhaustivos de atopar interacción ante a ausencia de efectos marxinais (pequenos efectos de asociación das variantes individuais que participan na epistasia). Por isto, esta tese continuou centrándose na optimización da busca exhaustiva de epistasia. Por unha parte, mellorouse a eficiencia do test de asociación a través dunha implantación vectorial do mesmo. Por outro lado, creouse un algoritmo distribuído que implementa unha busca exhaustiva capaz de atopar epistasia de calquera orden. Estes dous fitos lógranse en *Fiuncho*, unha ferramenta que integra toda a investigación realizada, obtendo un rendemento en clústeres de CPUs que supera a todas as súas alternativas no estado da arte. Adicionalmente, desenvolveuse unha libraría para simular escenarios biolóxicos con epistasia chamada *Toxo*. Esta libraría permite a simulación de epistasia seguindo modelos de interacción xenética existentes para orde alto.

Resumen

En los últimos años, los estudios de asociación del genoma completo (*Genome-Wide Association Studies*, GWAS) están ganando mucha popularidad de cara a buscar una explicación genética a la presencia o ausencia de ciertas enfermedades en los seres humanos. Existe un consenso entre estos estudios acerca de que muchas enfermedades complejas presentan interacciones entre los diferentes genes que intervienen en su expresión, un fenómeno conocido como epistasia. Esta tesis se centra en el estudio de este fenómeno empleando la computación de altas prestaciones (*High-Performance Computing*, HPC) y desde su perspectiva estadística: la desviación de la expresión de un fenotipo como suma de los efectos de múltiples variantes genéticas. Para ello se ha desarrollado una primera herramienta, *MPI3SNP*, que identifica interacciones de tres variantes a partir de un conjunto de datos de entrada. *MPI3SNP* implementa una búsqueda exhaustiva empleando un test de asociación basado en la Información Mutua, y explota los recursos de clústeres de CPUs o GPUs para acelerar la búsqueda. Con la ayuda de esta herramienta, hemos evaluado el estado del arte de la detección de epistasia a través de un estudio que compara el rendimiento de veintisiete herramientas. La conclusión más importante de esta comparativa es la incapacidad de los métodos no exhaustivos de localizar interacciones ante la ausencia de efectos marginales (pequeños efectos de asociación de variantes individuales pertenecientes a una relación epistática). Por ello, esta tesis continuó centrándose en la optimización de la búsqueda exhaustiva. Por un lado, se mejoró la eficiencia del test de asociación a través de una implementación vectorial del mismo. Por otra parte, se diseñó un algoritmo distribuido que implementa una búsqueda exhaustiva capaz de encontrar relaciones epistáticas de cualquier tamaño. Estos dos hitos se logran en *Fiuncho*, una herramienta que integra toda la investigación realizada, obteniendo un rendimiento en clústeres de CPUs que supera a todas sus alternativas del estado del arte. A mayores, también se ha desarrollado una librería para simular escenarios biológicos con epistasia llamada *Toxo*. Esta librería permite la simulación de epistasia siguiendo modelos de interacción existentes para orden alto.

Abstract

In recent years, Genome-Wide Association Studies (GWAS) have become more and more popular with the intent of finding a genetic explanation for the presence or absence of particular diseases in human studies. There is consensus about the presence of genetic interactions during the expression of complex diseases, a phenomenon called epistasis. This thesis focuses on the study of this phenomenon, employing High-Performance Computing (HPC) for this purpose and from a statistical definition of the problem: the deviation of the expression of a phenotype from the addition of the individual contributions of genetic variants. For this purpose, we first developed *MPI3SNP*, a program that identifies interactions of three variants from an input dataset. *MPI3SNP* implements an exhaustive search of epistasis using an association test based on the Mutual Information and exploits the resources of clusters of CPUs or GPUs to speed up the search. Then, we evaluated the state-of-the-art methods with the help of *MPI3SNP* in a study that compares the performance of twenty-seven tools. The most important conclusion of this study is the inability of non-exhaustive approaches to locate epistasis in the absence of marginal effects (small association effects of individual variants that partake in an epistasis interaction). For this reason, this thesis continued focusing on the optimization of the exhaustive search. First, we improved the efficiency of the association test through a vector implementation of this procedure. Then, we developed a distributed algorithm capable of locating epistasis interactions of any order. These two milestones were achieved in *Fiuncho*, a program that incorporates all the research carried out, obtaining the best performance in CPU clusters out of all the alternatives of the state-of-the-art. In addition, we also developed a library to simulate particular scenarios with epistasis called *Toxo*. This library allows for the simulation of epistasis that follows existing interaction models for high-order interactions.

Preface

With the proliferation of next-generation sequencing technologies, the cost of sequencing genomes has been reduced, and Genome-Wide Association Study (GWAS) have become more popular. GWAS are observational studies that attempt to decipher the relationship between a particular trait or phenotype and a group of genetic variants from several individuals. Much of the early work in GWAS considered genetic variants in isolation, and the results of those studies were unsatisfactory for the task at hand. The studies commonly reported associations with variants of unknown significance that increased disease risk at very low levels, and thus their usefulness in clinical applications was limited [1]. One hypothesis that explains this outcome is a phenomenon called epistasis: the interaction of genes among themselves, or with the environment, during the expression of a phenotype so that individual variants by themselves display little to no association with said phenotype. Nevertheless, looking for epistatic interactions instead of individually associated genetic markers is a much more complex task, and it is still an actively researched field.

This thesis is focused on the study of the epistasis phenomenon through High-Performance Computing (HPC) systems, and from its statistical definition: the deviation of the phenotype expression from the addition of the individual contributions of multiple genetic variants. We started by developing a program, *MPI3SNP*, that identifies epistatic interactions of three genetic variants from an input dataset containing many unrelated ones. *MPI3SNP* implements an association test based on the Mutual Information (MI) metric, and through the exploitation of the resources of a cluster of CPUs or GPUs, it performs an exhaustive analysis in a reduced amount of time compared to its alternatives. Using this program as a point of reference, we studied the state-of-the-art of the epistasis detection problem by comparing 27 different methods through empirical experimentation, with a special emphasis on high-order epis-

tasis. The most notable conclusion from this comparison is the limitation of non-exhaustive approaches to identify epistasis interactions when marginal effects are absent. Marginal effects refer to a small association effect that the individual genetic variants intervening in an epistasis interaction show in isolation. For this reason, this thesis continued improving the performance of the exhaustive search. On one hand, we proposed two vector implementation of the MI-based association test for two different vector widths of the *x86_64* CPU architecture: 256 and 512 bits. These two implementations support all modern *x86_64* microarchitectures. On the other hand, we developed a distributed algorithm for the detection of epistasis interactions of any given order. This algorithm makes use of the resources available in a CPU cluster to speed up the search. These two contributions are implemented in a program named *Fiuncho* that outperforms any other exhaustive alternative available in the state-of-the-art.

Outside of the epistasis detection problem, this thesis also contributed to the field of epistasis simulation. During the development of synthetic datasets in order to empirically study the different epistasis detection methods, we found that existing simulators were very limited when creating high-order epistasis interactions. For this reason, we created *Toxo*, a MATLAB library that compliments existing simulators by providing penetrance tables that allows them to simulate such epistasis interactions. Penetrance tables are tables that describe an epistasis interaction though the definition of the population frequency of the trait based on the different genotypes of the individuals, for all genetic variants intervening in the epistasis interaction.

Structure of the Thesis

The thesis is organized as follows:

- Chapter 1 introduces the thesis, defining what GWAS are, what is epistasis and the importance of its study. It also discusses the different approaches to the epistasis detection problem. To conclude, the chapter explains the association test used throughout the thesis that measures quantitatively the degree of association between a set of genetic variants and the trait under study.
- Chapter 2 discusses the design of a distributed algorithm dedicated to the exhaustive search of epistatic interactions of third order (comprised of three differ-

ent genetic variants). It also discusses the performance of its implementation, *MPI3SNP*, which exploits both CPU and GPU clusters.

- Chapter 3 presents a survey study comparing the state-of-the-art methods for high-order epistasis detection in terms of runtime, the ability to detect epistatic interactions, and the presence of false positives in their output.

- Chapter 4 studies the vector implementation of the association test introduced in Chapter 1. It describes how the algorithm can be implemented for the *x86_64* architecture through Advanced Vector Extensions (AVX) Intrinsics for two different vector widths, and compares the performance achieved with that of the autovectorization offered by modern compilers.

- Chapter 5 proposes a distributed algorithm for the detection of any-order epistasis interactions following an exhaustive search. The program, named *Fiuncho*, is inspired in the first algorithm presented in Chapter 2, and includes the vector implementations of the association test presented in Chapter 4. The chapter also empirically evaluates the performance of the program, and compares it to other epistasis detection software.

- Chapter 6 introduces *Toxo*, a library that compliments existing epistasis simulators. It calculates penetrance tables, tables that express the probability of having a particular trait based on the genotype information of the individuals, for high-order epistasis models. These tables can be provided to simulators to generate datasets that contain the epistatic interactions described by the model used.

- Chapter 7 summarizes the contributions made throughout the thesis, presents the conclusions reached and discusses the lines of future work that remain open.

- Appendix A documents the program configurations used during the survey presented in Chapter 2.

- Lastly, Appendix B includes an extended summary of this thesis written in Spanish.

Main contributions

The main contributions of this thesis are:

- A parallel CPU/GPU solution for exhaustive third-order epistasis detection [2].
- An extensive survey of high-order epistasis detection methods, with an experimental evaluation that compares the different programs in terms of elapsed time, epistasis detection power and presence of false-positives in the results [3].
- A parallel CPU application for exhaustive any-order epistasis detection that exploits all levels of parallelism of a homogeneous *x86_64* cluster [4, 5].
- A new method for the calculation of high-order penetrance tables for bivariate epistasis models [6, 7].

Developed Software

All software developed during this thesis is distributed as open source software via GitHub:

- *MPI3SNP*:

  https://github.com/UDC-GAC/mpi3snp

- *Fiuncho*:

  https://github.com/UDC-GAC/fiuncho

- *Toxo*:

  https://github.com/UDC-GAC/toxo

Publications

Journal Publications

- C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Fiuncho: a program for any-order epistasis detection in CPU clusters». In: *The Journal of Supercomputing* (2022)
- B. González-Seoane, C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «PyToxo: a Python tool for calculating penetrance tables of high-order epistasis models». In: *BMC Bioinformatics* 23.117 (2022)
- C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «A SIMD algorithm for the detection of epistatic interactions of any order». In: *Future Generation Computer Systems* 132 (2022), pp. 108–123
- C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Evaluation of existing methods for high-order epistasis detection». In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.2 (2022), pp. 912–926
- C. Ponte-Fernández, J. González-Domínguez, A. Carvajal-Rodríguez, and M. J. Martín. «Toxo: a library for calculating penetrance tables of high-order epistasis models». In: *BMC Bioinformatics* 21.138 (2020)
- C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Fast search of third-order epistatic interactions on CPU and GPU clusters». In: *The International Journal of High Performance Computing Applications* 34.1 (2020), pp. 20–29

Posters

- C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Poster: Genome wide association studies in high performance computing systems». In: *17th International Summer School on Advanced Computer Architecture and Compilation for High-performance Embedded Systems*. Fiuggi (Italy), 2021

Funding and Technical Means

This thesis was done thanks to the following funding programs and technical means:

- Working material, human and financial support primarily by the Computer Architecture Group of the Universidade da Coruña, along with a research fellowship funded by the Ministry of Education, Culture and Sports of Spain (FPU program, ref. FPU16/01333).
- Access to bibliographical material through the library of the Universidade da Coruña.
- Additional funding:
  - State funding by the Ministry of Economy and Competitiveness of Spain (project "Nuevos desafíos en computación de altas prestaciones: desde arquitecturas hasta aplicaciones (II)", ref. TIN2016-75845-P), the Ministry of Science and Innovation of Spain (project "Desafíos Actuales en HPC: Arquitecturas, Software y Aplicaciones", ref. PID2019-104184RB-I00 / AEI / 10.13039/501100011033)
  - Regional funding by the Galician Government (Xunta de Galicia) under the Singular Research Center of Galicia Program (ref. ED431G/01) and the Consolidation Program of Competitive Research Groups (refs. ED431C 2017/04, ED431G 2019/01 and ED431C 2021/30).
- Access to clusters and supercomputers:
  - *Pluton* cluster (Computer Architecture Group, Universidade da Coruña, Spain).
  - *Finisterrae-II* supercomputer (Centro de Supercomputación de Galicia, Spain).
  - *SCAYLE* supercomputer (Supercomputación Castilla y León, Spain).

Contents

List of Tables

List of Figures

List of Code Listings

List of Algorithms

List of Acronyms

ACO Ant Colony Optimization

API Application Programming Interface

AUC Area Under the ROC Curve

AVX Advanced Vector Extensions

BIC Bayesian Information Criterion

BMI Bit Manipulation Instructions

FMA Fused Multiply-Add

FWER Family-Wise Error Rate

GA Genetic Algorithm

GUI Graphical User Interface

GWAS Genome-Wide Association Study

HPC High-Performance Computing

MAF Minor Allele Frequency

MI Mutual Information

MPI Message Passing Interface

RR Round-Robin

SIMD Single Instruction Multiple Data

SNP Single Nucleotide Polymorphism

SPMD Single Program Multiple Data

SVM Support Vector Machine

SVML Short Vector Math Library

VPU Vector Processing Unit

Chapter 1

Introduction and Background

This chapter introduces some background concepts about Genome-Wide Association Studies (GWAS), epistasis and its detection, and contextualizes the research carried out in this thesis. It is structured into three sections: the first one, Section 1.1, defines GWAS; the second one, Section 1.2, explains the concept of epistasis and briefly summarizes the different approaches to its detection; and the last one, Section 1.3, presents a particular association detection method used in the upcoming chapters.

1.1. What Are GWAS?

GWAS are observational studies that try to decipher the relationship between a particular trait or phenotype and a group of genetic variants from several individuals. The association relationship is assessed through the differences in the allele frequencies of the variants between individuals that show different phenotype outcomes. A GWAS is a very lengthy process composed of many steps that begin with the design of the study and ideally conclude with the application of the newly found knowledge to disease risk prediction or a better understanding of the genetic architecture of the genome. Fig. 1.1 summarizes what steps are involved in a GWAS.

Since the publication of the first GWAS study more than a decade ago, many trait-associated variants have been reported. The GWAS catalog [14], a curated list of human GWAS studies, incorporates 325 538 associations from 5527 different publications as

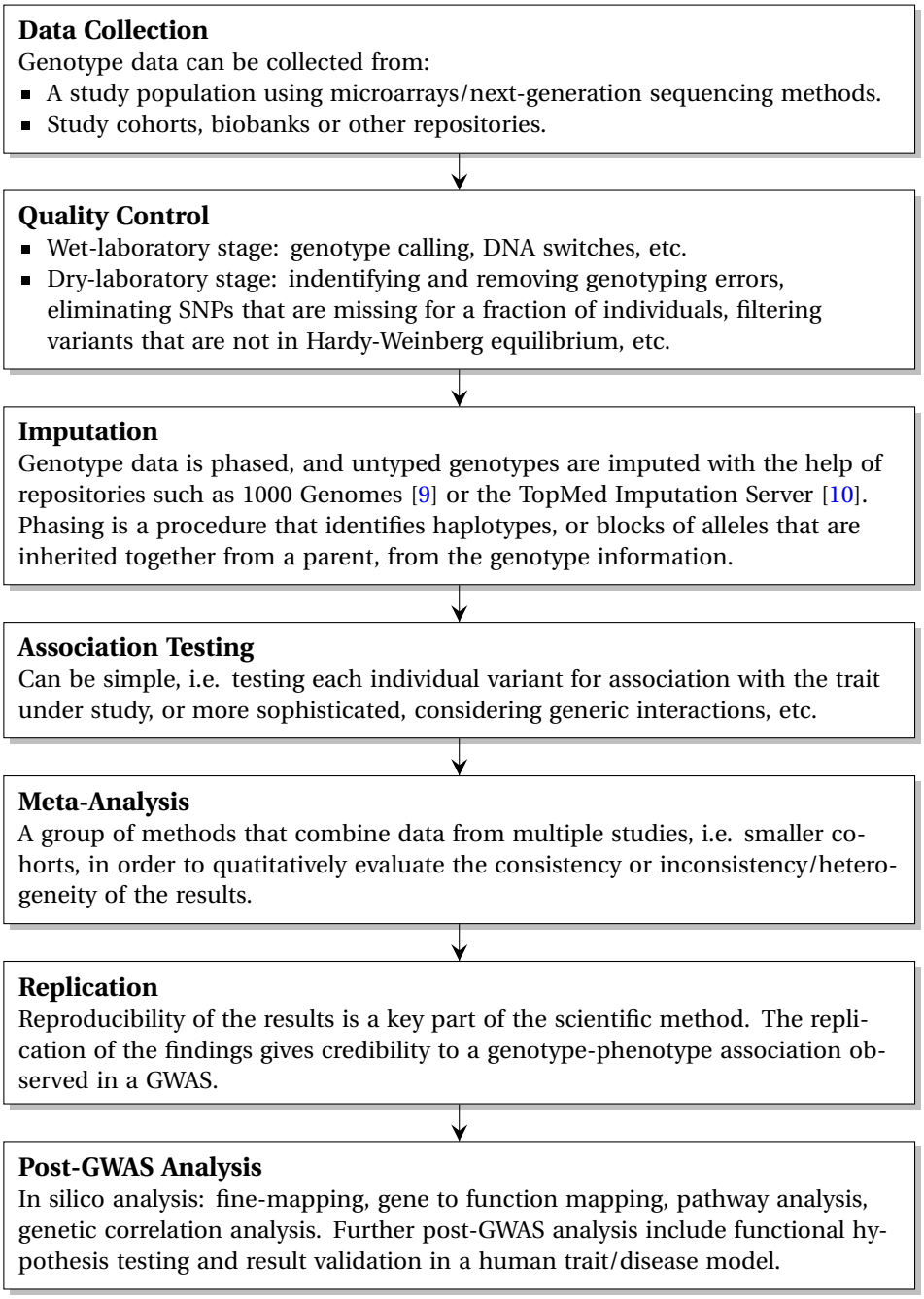


Figure 1.1: Flowchart of a typical GWAS, briefly describing the steps involved [11, 12, 13].

of December 2021. Despite the sheer number of results, these studies have not met the initial expectations, mostly consisting of variants of unknown significance that increase the disease risk at very low levels and are not very useful for clinic applications [15], and the significance of these findings was far from the heritability predicted from traditional genetic epidemiology studies [1]. Heritability refers to the amount of phenotype variation that can be explained due to genetic differences between individuals.

One hypothesis for the "missing heritability" problem is the presence of gene–gene and gene–environment interactions during the expression of a phenotype. These interactions are commonly known as epistasis.

1.2. What Is Epistasis?

This thesis focuses on the phenomenon of epistasis, the statistical interaction of genes among themselves, or with the environment, during the expression of a phenotype so that individual variants by themselves display little to no association with said phenotype. Epistasis was first introduced more than 100 years ago by William Bateson, and it is still an actively researched topic due to the computational challenge that it represents.

Identifying an epistasis interaction requires locating the combination of variants, or set of combinations, that best explains the phenotype under study. It is a combinatorial problem that grows with the number of variants and the size of the interaction (commonly referred to as the order) considered. For an epistasis search of k variants in combination from a total of n, there are $\binom{n}{k}$ candidate solutions to the problem.

Because of the dimension of the problem, epistasis detection authors use two different approaches: exhaustive and non-exhaustive methods.

1.2.1. Exhaustive methods

These are methods that examine every combination of variants available in the data, and locate the most associated ones with the phenotype under study. Fig. 1.2

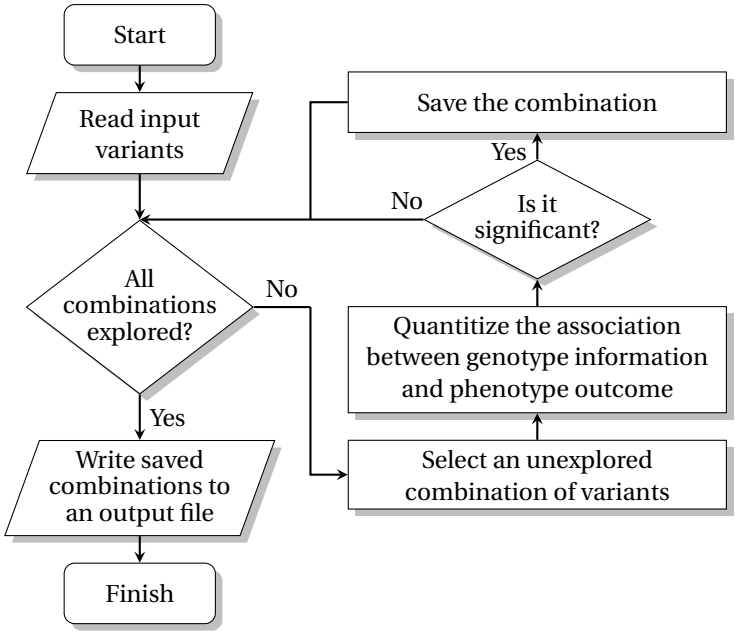


Figure 1.2: Flowchart of a typical exhaustive epistasis search.

shows a flowchart summarizing the process. As a consequence of that, all exhaustive methods present a computational complexity of $O(n^k \cdot A)$, with A being the computational complexity of each individual association test. The expression assumes that the number of combinations without repetition, $\binom{n}{k}$, is equivalent to n^k, since the epistasis order k is smaller than $n-k$.

This rigidity in the method itself has led to the development of proposals with more innovation in the different architectures used to tackle the problem than in the algorithmic approach to it. In the beginning, exhaustive methods did not target a computer architecture in particular. They were written in languages such as Fortran, Java or C, and could be used in any computer. This is the case of *MDR* [16], one of the most recognized exhaustive epistasis detection methods in the literature. *MDR* was written in Java, allows for epistasis interactions of any given order and supports multithreaded execution, although the performance achieved is not ideal in modern computers. Since then, performance has become the centerpiece of the exhaustive methods.

Currently, implementations are more tailored to a particular computer architecture in order to exploit all the resources offered to speed up the search. *BitEpi*, for

instance, uses an alternative representation of the genotype information in memory, introducing a tradeoff between the complexity of the association test and the use of a more memory-intensive approach to the computation. It implements a 2, 3 and 4-locus epistasis search that uses multithreading to speed up the search. Furthermore, for the *x86_64* CPU architecture, there are publications that discuss vector implementations of the epistasis search using Advanced Vector Extensions (AVX) [17].

Aside from CPUs, GPUs and FPGAs are two architectures that gained some interest from researchers in the field. GPUs are a great fit due to the high degree of parallelism that they offer and the embarrassingly parallel nature of the epistasis search. There are a multitude of methods that fall under this category, with *SNPInt-GPU* [18] being one of the latest examples. Furthermore, with the introduction of tensor cores in the most recent GPU microarchitectures, there has been effort to exploit these new instructions in the epistasis detection problem [19]. FPGAs have also been employed, with methods that support exhaustive 2 and 3-locus epistasis detection [20, 21], and more recently, epistasis interactions of any given order [22].

Lastly, some authors have embraced this diversity in architectures with methods that support heterogeneous systems in order to complete the epistasis search. This includes methods written in architecture-agnostic languages so that the same implementation can be compiled for different hardware [23], as well as methods that exploit computing systems with different architectures simultaneously, and thus taking advantage of the benefits of each separate architecture, such as CPUs with iGPUs [17], CPUs with GPUs [24] and GPUs with FPGAs [25].

However, to the best of our knowledge, only a preliminary study [26] with no real and available implementation considers cluster architectures for third or higher epistasis orders. *MPI3SNP* and *Fiuncho*, two programs developed during this thesis, address this limitation by exploiting the different resources offered by a cluster through the Message Passing Interface (MPI) library, in combination with other technologies.

1.2.2. Non-Exhaustive Methods

Non-exhaustive methods only test a fraction of the combinations of variants, following a particular strategy that reduces the search space. These methods reduce the

computational complexity at the cost of possibly missing the target variant combination. However, due to the reduced computational cost, they can be applied to larger problems and consider higher interaction orders.

Upton et al., in [27], categorizes non-exhaustive methods into two groups attending at the nature of the filtering procedure: data-driven and biological filtering. Data-driven filtering refers to the methods that apply some sort of quantitative measure to the genetic data in order reduce the number of tests performed, selecting a subgroup of individual variants or low-order combinations of them. These methods have a varying degree of complexity, going from simple methods that apply a statistic (such as the chi-squared test [28]) to the individual Single Nucleotide Polymorphisms (SNPs) in order to select a few, to more complex methods that apply concepts and algorithms from a multitude of fields: information theory [29, 30], machine learning [31, 32], regression analysis [33, 29], etc. As a result, there are no similarities common to all programs, and they display different computational complexities from one another. Chapter 3 presents a comprehensive review of methods that fall under the data-driven category, going further into the differences between detection strategies.

Biological filtering, on the other hand, uses biological knowledge to reduce the size of the epistasis search. Such approaches include the use of pathway databases, protein-protein interaction databases or genome-annotation based on function to reduce the search space [34]. There have also been comprehensive approaches that combine multiple sources of biological information to assist the search, such as *Biofilter* [35] or *INTERSNP* [36].

Non-exhaustive methods also make use of High-Performance Computing (HPC) architectures to speed up the search [37, 30], although these are less frequent than exhaustive methods due to the less time-consuming nature of the non-exhaustive approach.

1.3. A Mutual-Information-Based Association Test

All epistasis detection methods need an association statistic to quantify the degree of association between a particular combination of variants and the phenotype in question. Not every metric is appropriate to find this correlation. Chi-squared tests,

frequently used for pairwise interactions, are inaccurate when low genotype frequencies are present, which become increasingly common in high-order interactions as not every genotype combination includes a significant fraction of the population. Mutual Information (MI), as stated by Leem et al. in [38], is adequate for this purpose, that is why we will use an association test based on MI in this thesis. This test has been compared against alternative association metrics in the review presented in Chapter 2 and showed exceptional detection power along other methods.

The MI test can be divided into three different steps: the construction of the genotype and contingency tables of the variant combination in order to obtain the frequencies of the different genotypes segmented by the phenotype, and the usage of the MI statistic to quantify the association between the genotype frequencies and the phenotype.

The computational complexity of this association test is $O(3^k m)$, with k being the number of variants in combination tested, and m the number of samples per variant.

1.3.1. Constructing the Genotype Table

Genotype tables represent, in binary format, the genotype information of all individuals under study for a particular variant or combination of variants. They are a generalization of the binary representation introduced in *BOOST* [39] to simplify the computation of contingency tables for epistasis interactions of second order. The tables contain as many columns as individuals in the data, segregated into cases and controls, and as many rows as genotype values a variant or combination of variants can show. Every individual has a value of 1 in the row corresponding to its genotype, and a 0 in every other row. For a human population with biallelic markers, each individual can have three different genotypes, and thus genotypes tables contain 3^k rows with k being the number of variants in combination represented.

Genotype tables not only are used to represent the information of a variant, but also to segment the individuals into different groups by their phenotype and genotype values and to represent the information of multiple variants in combination. This makes them extremely useful later when computing the frequencies of each genotype value. The construction of a genotype table for a combination of multiple variants implies:

Genotype table of variant *a*:

	Cases	*Controls*
a_1	00101101	10101001
a_2	10000000	01000010
a_3	01010010	00010100

Genotype table of variant *b*:

	Cases	*Controls*
b_1	11010000	00010101
b_2	00000110	00100010
b_3	00101001	11001000

Genotype table of variant *a* × variant *b*:

	Cases	*Controls*
$a_1 \cap b_1$	00000000	00000001
$a_1 \cap b_2$	00000100	00100000
$a_1 \cap b_3$	00101001	10001000
$a_2 \cap b_1$	10000000	00000000
$a_2 \cap b_2$	00000000	00000010
$a_2 \cap b_3$	00000000	01000000
$a_3 \cap b_1$	01010000	00010100
$a_3 \cap b_2$	00000010	00000000
$a_3 \cap b_3$	00000000	00000000

Figure 1.3: Example of two genotype tables of two different variants, a and b, for eight cases and controls, and the combined genotype table of the two variants.

a) the combination of the different rows of the tables corresponding to the individual variants, and

b) the computation of the intersection of each combination of rows (or genotype groups) via bitwise *AND* operations.

Fig. 1.3 shows an example of two genotype tables for two variants a and b for sixteen individuals (eight cases and controls), and the table resulting from the combination of these two variants.

1.3.2. Obtaining the Contingency Table

A contingency table is a type of table that holds the frequency distribution of a number of variables, that is, the genotype and phenotype distributions for this domain of application. These frequencies can be directly obtained by counting the number of individuals in each of the phenotype and genotype groups created by the genotype table. This implies counting the number of bits set, an operation commonly known as a population count. Fig 1.4 shows the contingency tables of the example genotype tables included in Fig. 1.3.

Contingency table of variant *a*:

	Cases	*Controls*
a_1	4	4
a_2	1	2
a_3	3	2

Contingency table of variant *b*:

	Cases	*Controls*
b_1	3	3
b_2	2	2
b_3	3	3

Contingency table of variant *a* × variant *b*:

	Cases	*Controls*
$a_1 \cap b_1$	0	1
$a_1 \cap b_2$	1	1
$a_1 \cap b_3$	3	2
$a_2 \cap b_1$	1	0
$a_2 \cap b_2$	0	1
$a_2 \cap b_3$	0	1
$a_3 \cap b_1$	2	2
$a_3 \cap b_2$	1	0
$a_3 \cap b_3$	0	0

Figure 1.4: Contingency table examples using the same variants as in Fig. 1.3.

1.3.3. The Mutual Information Metric

Once the contingency table is calculated, the only step left to assess the association between the genotype distribution and the phenotype affliction is computing the MI of the table. Considering two random variables X and Y representing the genotype and phenotype variability, respectively, the MI can be obtained as:

$$MI(X;Y) = H(X) + H(Y) - H(X,Y) \tag{1.1}$$

where $H(X)$ and $H(Y)$ are the marginal entropies of the two variables, and $H(X,Y)$ is the joint entropy. Marginal entropies of one and two variables are obtained as:

$$H(X) = -\sum_{x \in X} p(x) \log p(x) \tag{1.2}$$

$$H(X,Y) = -\sum_{x,y} p(x,y) \log p(x,y) \tag{1.3}$$

with $p(x)$ representing the probability of the random variable X taking the value x, $p(y)$ the probability of the random variable Y taking the value y, and $p(x,y)$ the

joint probability of both events. These probabilities can be obtained directly from the contingency table as the division between the number of occurrences and the number of total observations.

Chapter 2

Third-Order Epistasis Search on CPU and GPU Clusters

This chapter discusses the design of a distributed algorithm dedicated to the exhaustive search of third-order epistasis interactions, implemented in *MPI3SNP* [2], that supports both CPU and GPU clusters. The chapter is organized as follows. Section 2.1 describes the parallel implementation proposed and how the work is scheduled among the architecture to obtain a good load balance. Section 2.2 shows an experimental evaluation of *MPI3SNP*'s parallel efficiency. Finally, Section 2.3 draws some conclusions and offers some lines of future work.

2.1. MPI3SNP

Exhaustively searching third-order combinations, using the association test presented in Section 1.3, has a computational complexity of $O(n^3m)$, with n being the number of input SNPs and m the number of individuals considered. With the hope of mitigating the inherent cubic time complexity of the problem, and thus to be able to deal with relatively large datasets, in a previous work, González-Domínguez et al. proposed *GPU3SNP* [40], a tool that is able to exploit several GPUs within the same node to exhaustively search third order epistatic interactions. Although the results show that *GPU3SNP* achieves high performance and significantly reduces the execution times,

the analysis of large GWAS datasets would still require a significant amount of time. For instance, the analysis of a dataset consisting of 50 000 SNPs and 1000 individuals needs nearly 22 hours on a computing node with 4 NVIDIA GTX Titan GPUs. Thus, for large datasets, HPC cluster architectures should be used instead. *MPI3SNP* is designed to exploit cluster architectures. It is a hybrid two-level parallelization approach supporting nodes with both multicore CPUs and GPUs. On the inter-node level, MPI is used to communicate different nodes for both implementations. On the intra-node level, *Pthreads* and CUDA are used for the CPU and GPU implementations, respectively.

2.1.1. Distribution Strategy

In the exhaustive epistasis search, the task that consumes the runtime of the program almost entirely is measuring the degree of association of all variant combinations. Each association test involves the same computations, and thus the combinations themselves implicitly divide the problem into smaller tasks of similar weight. However, many of the 3-SNP combinations share sub-combinations with one another, and as such, many repeating computations concerning the construction of the genotype tables (see Section 1.3.1) can be avoided attending to how the combinations are scheduled on the different computing units (in this scenario, they can be CPU cores or GPU cards). For instance, the analysis of the combinations with variants {1,2,3}, {1,2,4}, {1,2,5}, etc. requires the construction of the same genotype table corresponding to the pair {1,2}. One solution to this problem is to assign all these combinations to the same unit, which enables the reuse of the genotype table of the SNPs {1,2} for all third-order combinations that contain them.

Following this strategy, pairs are assigned using a Round-Robin (RR) distribution among all computing units. Each unit computes, for every pair assigned, all non-repeating combinations of three SNPs that begin with that pair, maximizing the reuse of operations. Although the number of possible combinations varies from pair to pair, the cyclic distribution guarantees a good balance. Fig. 2.1 gives an example of this approach, showing the distribution of all the triplets resulting from 9 SNPs among 3 computing units. This example uses a very small number of SNPs and some differences in the number of combinations assigned per unit can be noted. However, with a more realistic input size, the differences are unnoticeable in relation with the total number

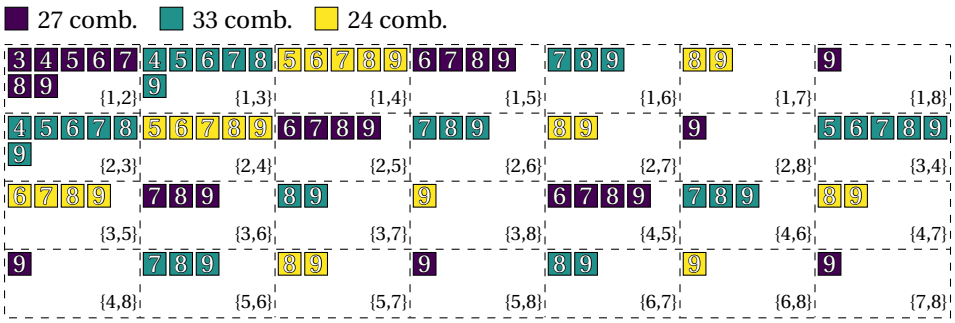


Figure 2.1: Example of the distribution strategy, arranging combinations of three SNPs among three computing units. Each prefix of two SNPs (represented as rectangles with dashed lines) is assigned to a unit (shown with different colors) following a RR distribution, and that unit tests for association every combination of three SNPs starting with the prefix (represented as small colored squares).

of assigned combinations, both in an intra-node and inter-node level, as it will be seen in the posterior experimental evaluation (Section 3.2).

This strategy adheres to the Single Program Multiple Data (SPMD) paradigm in which all computing units execute the same function, while each unit analyzes a different set of variant combinations. It does not require communication among computing units during the computation step. This is achieved by replicating the input data among all computing units (avoiding the need of communications among processes when constructing the contingency tables) and gathering all results intro a single unit at the end of the computation. The consequent memory overhead due to replication is affordable on current multicore clusters. For instance, storing the genotype data of the biggest dataset employed in the experimental evaluation (6300 SNPs and 3200 samples) only requires 58 MB of memory.

Since there are no communications between the start and end of the program, the distribution strategy can be completely abstracted from the topology of the hardware. The distribution assigns combinations to the computing units directly, which are CPU threads in a CPU execution or GPU cards during a GPU execution.

2.1.2. CPU Clusters

The CPU implementation combines MPI multiprocessing with multithreading to efficiently exploit the computational capabilities of CPU clusters. Each computing unit corresponds to each CPU core partaking in the computation. Every MPI process reads the input SNPs and stores each one in a genotype table, keeping the individual variant information replicated in each process. After that, each MPI process spawns a number of threads that execute the MI association test over a different set of variant combinations. The input data is provided to the different threads through shared memory, making an efficient use of the memory inside each node. Once the combinatory exploration of all threads from a process conclude, the results are agregated into a single list of combinations, then sorted by their MI value and truncated to the number of desired outputs. Analogously, when all processes conclude, their results are gathered into the process with id 0, sorted by their MI, truncated and written into an output file.

2.1.3. GPU Clusters

On GPU clusters, a computing unit in the distribution correspond to a GPU device. Each MPI process controls a single GPU, transferring its fraction of the workload to the GPU and calling the appropriate compute kernels (written in CUDA C++) to process its corresponding triplets.

In order to minimize the thread divergence in the GPUs, the workflow is divided into two kernels. The first one calculates the genotype tables of the assigned SNP pairs, and stores these partial results into the device memory. The second kernel uses these partial results to calculate the MI for all the triplets that can be generated with those pairs. Since GPU memory is limited, to be able to store all the partial results, the workload is transferred from the MPI process to the GPU in blocks, following an iterative procedure until its fraction of the computation is completed. MI results are kept in the GPU memory until the end, and then gathered into a single MPI process.

As explained during the introduction in Chapter 1, *MPI3SNP* uses the genotype table representation to accelerate the contingency table construction step. In the CPU version, all the data is consecutively stored by SNP to exploit cache locality. However, due to differences in the memory hierarchy, a data transposition is applied in the GPU

Table 2.1: Hardware characteristics of the *Finisterrae-II* cluster.

	CPU NODE	K80 NODE	K2 NODE
OS	RHEL 7.5	RHEL 7.5	RHEL 7.5
CPUS	2x Intel E5-2680 v3 (24C)	2x Intel E5-2680 v3 (24C)	2x Intel E5-2650 v3 (20C)
MEMORY	128 GB	128 GB	128 GB
GPU CARDS	-	2x NVIDIA K80	NVIDIA K2
NETWORK	Infiniband FDR@56Gbps	Infiniband FDR@56Gbps	Infiniband FDR@56Gbps

version in order to increase the coalescence and thus the performance of the device memory accesses. More details about the data transposition and its benefits, as well as the CUDA kernels and the iterative SNP block transfer can be found in the original work of [40].

2.2. Experimental Evaluation

The experimental evaluation consists of two parts: a parallel balance evaluation, which measures the differences in assigned workload among computing units, and a speedup and efficiency evaluation in which the runtimes of the application, both for CPU and GPU executions, are compared. The epistasis detection power and false positive rate are extensively evaluated in the following Chapter 3, showing very good results compared to other state-of-the-art methods.

The evaluation was performed on the *Finisterrae-II* cluster, described in Table 2.1. As per the software environment, GCC 5.3, NVCC 8.0 and *OpenMPI* 1.10 were used. The datasets used in all executions consist of 6300, 5000, 4000 and 3200 SNPs, with 3200 samples (1600 cases and 1600 controls) per SNP. These datasets were obtained from [40] by trimming the number of samples and SNPs.

2.2.1. Parallel Distribution Balance

Fig. 2.2 evaluates the workload balancing strategy using a fixed problem size of 3200 SNPs (and thus a fixed number of combinations to distribute) and a growing number of computing units from 12 to 768. The measure used is the relative differences (in percentage) of assigned combinations from the mean number of combinations per

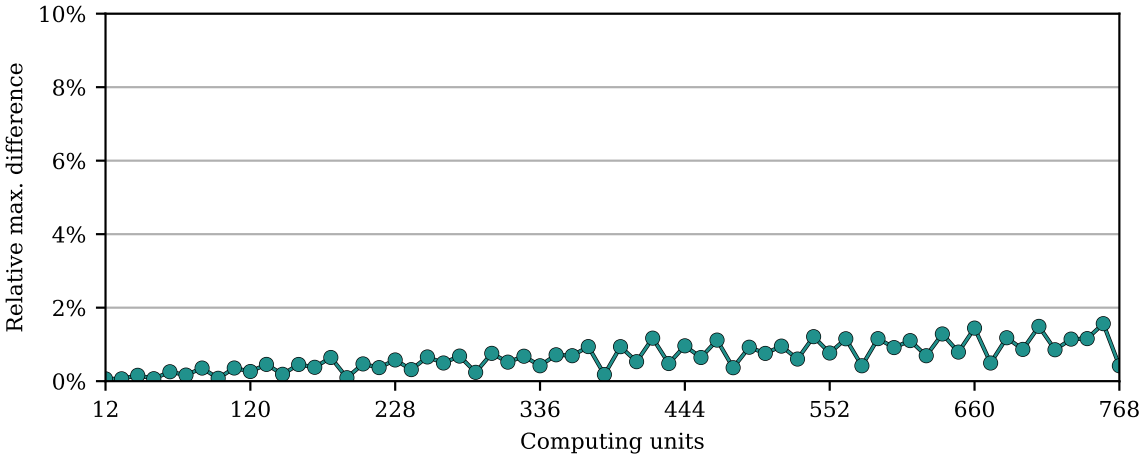


Figure 2.2: Maximum difference of assigned combinations to any computing unit from the average number of combinations assigned per unit, relative to the latter.

unit (which represents the most balanced distribution attainable), calculated as:

$$100\,\frac{\max d_i - \binom{n}{3}/p}{\binom{n}{3}/p} \tag{2.1}$$

with d_i being the number of combinations assigned to the unit i, n the number of variants and p the number of computing units used. The figure shows that, for every scenario tested, the difference is under 2 %. For scenarios with a larger variant count, as is the case during the following experimental evaluation, the differences in assigned workload are expected to be even smaller.

2.2.2. CPU Speedup and Efficiency

The CPU evaluation distinguishes between intra-node and inter-node scenarios. For the intra-node evaluation, two different configurations are considered: a single-thread execution and a 24-thread execution. For the inter-node evaluation, the performance of a single-node execution is compared with the performance of 4, 8, 16 and 32 nodes.

Table 2.2: Sequential CPU runtimes of *MPI3SNP*, for datasets consisting of 6300, 5000, 4000 and 3200 SNPs, with 3200 per SNP.

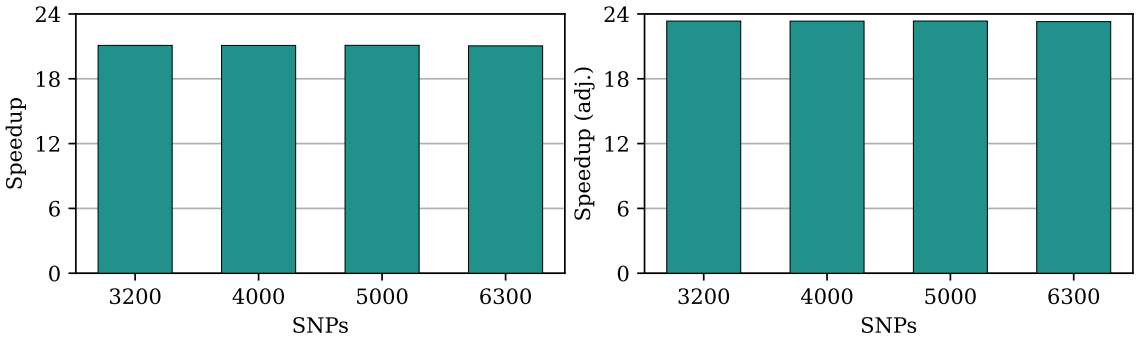
SNPs	Elapsed time (s)
3200	29323
4000	57274
5000	111887
6300	224132

Intra-node evaluation

Fig. 2.3 shows the speedups obtained from a multithread execution using the 24 cores available in a single node for different input sizes, compared to the sequential runtimes using a single core shown in Table 2.2. The figure distinguishes between observed and frequency-adjusted speedups. The observed speedup is calculated as T_1/T_N, with T_1 being the elapsed time using a single CPU core and T_N the elapsed time using N CPU cores. This metric is far from the ideal efficiency, and this is due to the frequency scaling present in modern processors. Intel CPUs, in particular, adjust their maximum core frequencies attending to the number of active cores, among other factors [41]. Therefore, to get a better grasp of the efficiency of the parallel implementation, an adjusted speedup compensating for the discrepancy in average CPU frequency is included in the figure, calculated as $T_1/T_N \cdot F_1/F_N$, where F_1 is the average single-core frequency and F_N is the average multicore frequency when N cores are used. The results for a single-node (24 cores) execution show very good efficiencies when the speedup is adjusted for the frequency differences between single-core and multicore executions.

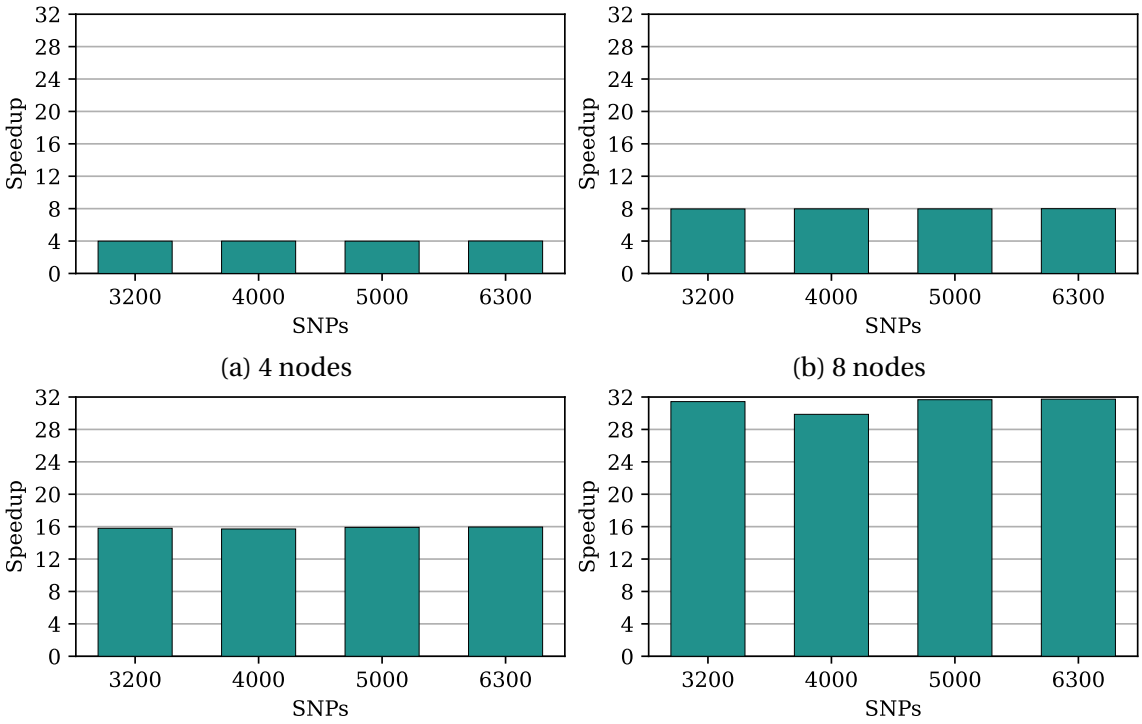
Inter-node evaluation

Fig. 2.4 compares the speedups achieved as the number of nodes increases for the same dataset sizes. The inter-node evaluation does not present the previous frequency differences, as every node is using all cores during the execution. In this scenario, the speedup is calculated as T_1/T_N, with T_1 being the elapsed time using a single node (24 cores) and T_N the elapsed time using N nodes (with $24 \times N$ cores). The speedups obtained in all scenarios show a parallel efficiency close to one.



(a) Observed speedup (b) Adjusted speedup

Figure 2.3: Speedups of *MPI3SNP* for multithread executions using 24 threads compared to the single-thread results from Table 2.2, representing both the observed and frequency-adjusted speedups.

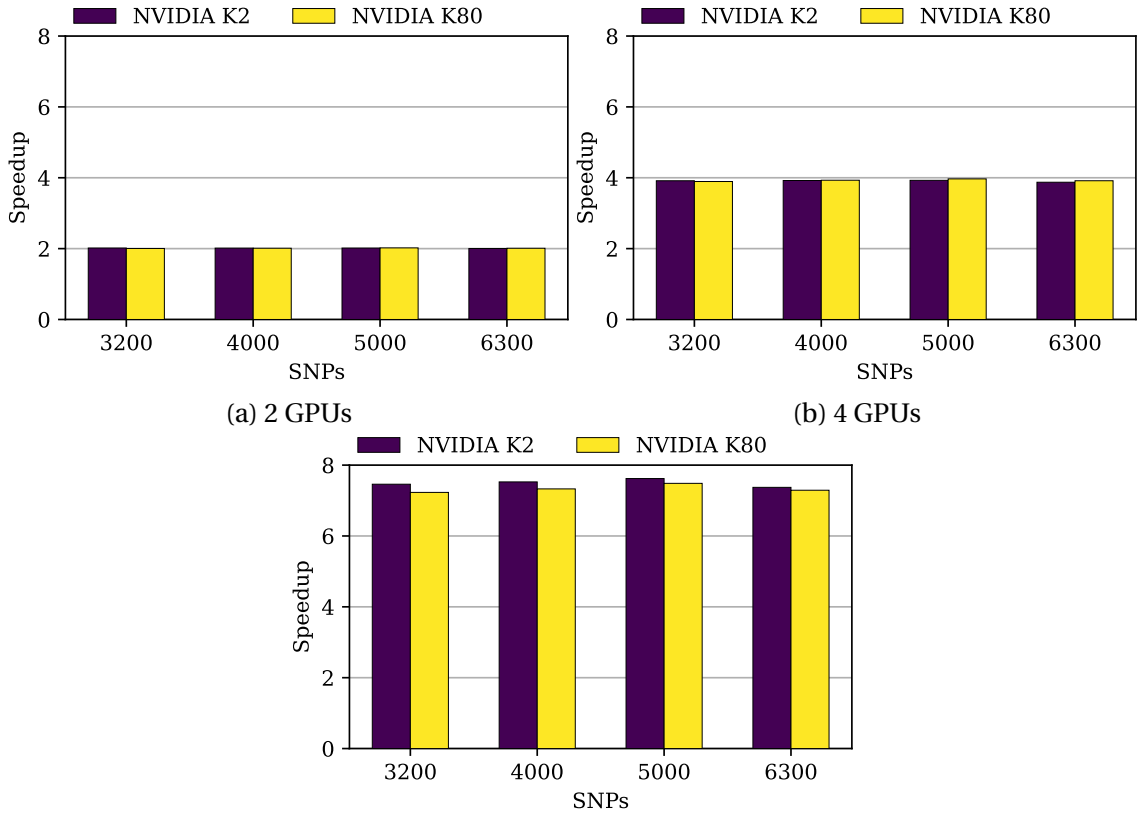


(a) 4 nodes (b) 8 nodes

(c) 16 nodes (d) 32 nodes

Figure 2.4: Speedups of *MPI3SNP* using 4, 8, 16 and 32 nodes with 24 threads per node, compared to a single-node execution.

To give some perspective, remark that the sequential runtime for the dataset with 6300 SNPs and 3200 individuals is roughly 62 hours. The same epistasis search is re-



(a) 2 GPUs

(b) 4 GPUs

(c) 8 GPUs

Figure 2.5: Speedups using a variable number of GPUs, compared to the single-GPU executions from Table 2.3.

duced to under 6 minutes by exploiting 32 CPU nodes of the cluster.

2.2.3. GPU Speedup and Efficiency

Fig. 2.5 compares the speedup obtained from using multiple GPUs compared to the runtimes of a single GPU execution for different dataset sizes, shown in Table 2.3. This study uses the NVIDIA K2 and NVIDIA K80 GPUs (Table 2.1), available in four and two nodes of the *Finisterrae-II* cluster. It is worth mentioning that these are dual GPU cards, making a total of eight GPU cards from each type. Single GPU results show that the NVIDIA K80 GPU is considerably faster than the K2. The speedups obtained with the two different GPUs can be assumed as linear for 2 and 4 GPUs, and shows slightly degraded efficiencies for 8.

Table 2.3: Runtime of *MPI3SNP* on a single GPU, for datasets consisting of 6300, 5000, 4000 and 3200 SNPs, with 3200 per SNP.

SNPs	NVIDIA K2 (s)	NVIDIA K80 (s)
3200	748	241
4000	1459	469
5000	2866	919
6300	5519	1775

The thousands of cores available in a GPU make it adequate to accelerate applications that are highly parallelizable. This is the case for epistasis detection. For instance, using the largest dataset size (6300 SNPs and 3200 samples), eight GPUs reduce the runtime from approximately 92 and 30 minutes for the K2 and K80 respectively, to 12 and 4 minutes.

2.3. Concluding Remarks

The principal limiting factor of epistatic searches is the exponential growth of the number of combinations with the number of SNPs involved on the interaction. Instead of reducing the scope of the search by means of a previous filtering stage, or limiting the usability to small datasets, our approach relies on cluster architectures to overcome this exponential growth and achieve a manageable run-time. The current implementation obtains next to linear speedups thanks to the use of a static distribution of the workload that allows the reuse of computations, avoids communications and synchronizations, and provides an almost perfect load balance.

Take as an example the largest dataset tested, composed by 3200 samples of 6300 SNPs each. When using 32 nodes of 24 CPU cores each, the obtained speedup is 31.73 compared to a single-node execution. The GPU implementation shows similar parallel efficiencies, as the total runtime is reduced 7.29 times when using 8 NVIDIA K80 GPUs against one.

In spite of the highly scalable parallel application, the exponential computational complexity of the exhaustive search could make its extension to higher interaction orders unfeasible for analyzing many SNPs. For this reason, the next chapter compares the available methods in the literature to gauge which one is the best alternative to

extend the epistasis search to any-order interactions.

Chapter 3

Review of High-Order Epistasis Detection Methods

In this chapter, we compare the epistasis detection methods published between 2009 and 2019, with a special interest in high-order epistasis detection. The chapter is structured as follows. Section 3.1 provides a brief description of the methods included in the study. Section 3.2 evaluates all methods in terms of runtime, detection power and Family-Wise Error Rate (FWER). Section 3.3 discusses the results and summarizes the findings. At last, Section 3.4 completes the chapter with the conclusions.

3.1. Methods

Prior to this survey, there have been several review studies that compared different strategies for epistasis detection from various perspectives. Some are focused entirely on their methodology, comparing the different approaches, their advantages and limitations [42, 43, 44, 27, 45, 46, 47]. Other studies go further by also including an empirical comparison from simulation studies, although the number of methods included in these studies is more limited [48, 49, 50, 51, 52]. There are also previous publications regarding the selection of epistatic detection methods and how to integrate them in the different stages of a genetic study [53, 54, 55]. Nevertheless, there is no previous comparison study with an emphasis on the interaction order.

Table 3.1: List of the different methods evaluated, together with the strategy followed, the implementation language used and the year of publication.

METHOD	STRATEGY	LANGUAGE	YEAR	REF.
AntMiner	Swarm intelligence	MATLAB	2012	[56]
ATHENA	Genetic Algorithm	C++	2010	[31]
BADTrees	Depth-first	C++	2012	[57]
BEAM3	Random-search based	C++	2012	[58]
BHIT	Random-search based	C++	2015	[59]
CINOEDV	Swarm intelligence	R	2016	[60]
DCHE	Filtering	Java	2014	[37]
EACO	Swarm intelligence	MATLAB	2018	[61]
EDCF	Filtering	C/C++	2012	[62]
epiACO	Swarm intelligence	MATLAB	2017	[63]
EpiMiner	Filtering	MATLAB	2014	[29]
FDHE-IW	Depth-first	MATLAB	2018	[64]
GALE	Genetic Algorithm	Python	2010	[65]
HiSeeker	Filtering	C++	2017	[66]
IACO	Swarm intelligence	MATLAB	2016	[67]
LAMPLINK	Filtering	C++	2016	[68]
LRMW	Depth-first	C++	2014	[69]
MACOED	Swarm intelligence	C++/MATLAB	2015	[32]
MDR	Exhaustive	Java	2001	[70]
MECPM	Filtering	C	2009	[71]
Mendel	Filtering	C/C++	2009	[72]
MPI3SNP	Exhaustive	C++	2019	[2]
NHSA-DHSC	Swarm intelligence	MATLAB	2017	[73]
SingleMI	Filtering	C++ & CUDA	2017	[30]
SNPHarvester	Random-search based	Java	2009	[74]
SNPRuler	Depth-first	Java	2010	[75]
StepPlr	Depth-first	R	2008	[33]

This review includes the methods that, first, support epistasis detection for qualitative phenotypes and for more than two variants; second, offer an implementation freely available to the scientific community and finally, their execution can be completed within a week. Table 3.1 lists all methods included. We decided to also consider *MDR* and *StepPLR*, despite being published more than ten years ago, due to their relevance in the field.

The selected methods have been grouped into six categories, attending at how the search space is explored: exhaustive methods, filtering methods, depth-first methods, swarm intelligent methods, genetic algorithms and random-search-based methods. Here, we provide a brief description of them in order to highlight the similarities and

differences between methods, and to have a better understanding of the results that each program yields. We refer to the authors' original works for a more complete and in-depth explanation.

3.1.1. Exhaustive Methods

As commented in Chapter 1, exhaustive methods apply the brute force technique to the association search problem, exploring all possible combinations of genetic markers up to a defined size or order. The computational cost of exploring all possible combinations is exponential with the number of genetic markers considered and the combination size. Therefore, these methods cannot be applied to large datasets with high epistatic factors.

MDR [70] and *MPI3SNP* [2] fall under this category. *MDR* partitions the individuals in the dataset into different k-fold cross-validation groups. Combinations are evaluated through a prediction model which labels the different allele combinations as high-risk (if the number of cases exceeds the number of controls for that particular combination) or as low-risk (if it does not). For each combination, k different models are created (one per cross-validation partition) and its prediction accuracy is averaged across partitions. At the end of *MDR*, the combination corresponding to its best-averaged prediction accuracy is reported. *MPI3SNP*, as explained in the previous chapter, enumerates all third-order combinations and sorts them using MI, returning the top-ranked ones. However, in this chapter we use a modified version of the program to also support fourth-order combinations.

3.1.2. Filtering Methods

Filtering methods discard many SNPs or combinations of SNPs to reduce the computational burden. The most direct approach is to filter the individual SNPs of the dataset before attempting to combine them, drastically decreasing the number of combinations. *EpiMiner* [29] and *Mendel* [72] follow this approach. *EpiMiner* ranks individual SNPs by their co-information index and retains the top ranked ones. The number of retained SNPs can be fixed or selected on a case-by-case basis through a Support Vector Machine (SVM). The retained SNPs advance to a second stage where all possible

combinations among them are evaluated using permutation-based co-information, and combinations whose p-values surpass a certain threshold are reported as interactions. Computing the co-information index requires calculating the index for all the combinations which contain a certain SNP up to a certain order, which still supposes a costly step, therefore *EpiMiner* allows us to approximate its value through Monte Carlo sampling. *Mendel* uses a lasso penalized logistic regression model to quantify the association between the SNPs, used as predictor variables, and the phenotype, used as the regression class. The interaction search process begins by pre-screening the SNPs in the dataset in a first stage using a simplified regression model and an absolute score criterion. Then, the number of SNPs selected is further reduced by tuning the constant λ, which increases the lasso penalty and, in turn, leaves many predictors out of the logistic regression model. Finally, when the number of retained SNPs is very small, the penalty is removed and the model coefficients are re-estimated. Using this final model, p-values of individual and combinations of SNPs are assessed following a leave-one-out procedure and thus the associated combinations are identified.

Alternatively, other methods perform the filtering step on low-order combinations. *HiSeeker* [66] and *MECPM* [71] enumerate all possible 2-SNP combinations and select a group of candidates for further analysis. *HiSeeker* filters these combinations by applying Pearson's χ^2 test with eight degrees of freedom, assessing the association between each combination and the phenotype. Combinations that meet a relaxed Bonferroni-corrected p-value threshold proceed to a second stage for a higher-order analysis. *HiSeeker* offers the possibility of performing an exhaustive search during the second stage to find high-order interactions, or using an Ant Colony Optimization (ACO) algorithm if the number of combinations to be tested is still unreasonably high. ACO algorithms will be covered in detail in Section 3.1.4. In the end, the non-relaxed Bonferroni-corrected p-value threshold is used to filter false positives. *MECPM* creates a maximum entropy classification model and uses the Bayesian Information Criterion (BIC) to quantify the association between genotypes and the phenotype under study. For this purpose, *MECPM* first creates a pool of promising SNP combinations and iteratively adds combinations to the model until the BIC cost is minimum. The pool is constructed following two approaches: a complete approach where all single SNPs and combinations of two SNPs serve as seeds, and successive SNPs are appended to each seed measuring the change in BIC cost with each addition; and a greedy approach where the initial selected seeds are reduced to the top-ranking sin-

gle and combinations of two SNPs using the relative entropy, and successive SNPs are appended maximizing this metric. *MECPM* reports the SNP combinations included in the model.

DCHE [37], *EDCF* [62] and *SingleMI* [30] use clustering techniques to filter combinations of SNPs. Both *DCHE* and *EDCF* recursively apply a clustering algorithm over the population frequencies of all allele combinations, starting from 2-SNP combinations up to a selected order. These clusters are then tested using Pearson's χ^2 test to measure its association with the phenotype. *DCHE* implements a clustering algorithm named *Dynamic Clustering* which reduces the 3^k frequencies associated with a combination of k SNPs in a biallelic population to a number between 3 and 6, merging the two least significant allele combinations in each step. *DCHE* retains a different fixed number of top-ranking combinations depending on the combination order being explored and applies a p-value threshold at the end of the algorithm to filter out irrelevant combinations. *EDCF*, instead, creates three groups from all allele combination frequencies: G_0, or combinations which occur more frequently in cases than in controls; G_1, or combinations which occur more frequently in controls than in cases; and G_2 with the combinations left. Clusters are then evaluated using a permutation test and the corresponding SNP combination is discarded if their p-value does not meet a certain threshold. Again, a fixed number of top-ranking SNP combinations (using the aforementioned χ^2 test) are retained from each combination size and its Bonferroni-corrected p-value is finally used as the threshold to decide the result of the method. *SingleMI* uses a clustering algorithm in a very different manner from the previous two. Individual SNPs are clustered following a k-means clustering method, where the distance between SNPs and the centroid of each cluster is measured using MI. Markers that are strongly interacting pair-wise tend to be placed in different clusters. Therefore, after creating the K clusters, a user-defined number of SNPs from different clusters are analyzed exhaustively using the same MI metric.

LAMPLINK [68] follows a completely different filtering approach from previous methods. Individual SNP genotypes are first categorized into two classes following a dominant or recessive exclusive model: risk and non-risk classes. Then, a modified version of the pattern mining algorithm called Linear time Closed itemset Miner [76] is used to prune the SNPs combinations that, taking into account their frequency, cannot show a significant association with the phenotype. Finally, the non-pruned combina-

tions are evaluated using a Fisher's exact test or a chi-squared test and the obtained p-value is corrected according to the number of testable combinations.

3.1.3. Depth-First Methods

This group is made of methods that explore the combination space using a depth-first search method, incorporating SNPs on each iteration while maximizing some measurement until convergence is detected. This search is repeated successively until a certain number of combinations is reached or no more significant combinations can be found. *FDHE-IW* [64], *LRMW* [69], *BADTrees* [57], *StepPLR* [33] and *SNPRuler* [75] follow this procedure.

FDHE-IW implements a search algorithm which constructs SNP combinations incrementally, starting with the empty set and repeatedly adding the SNP that maximizes the Symmetrical Uncertainty of the set until a maximum set size is reached. A G-test is applied after achieving a number of combinations to obtain a p-value associated with the combinations. *LRMW* uses decision trees to represent candidate interactions and employs its associated Area Under the ROC Curve (AUC) to measure significance. The method starts with an empty tree and progressively generates more complex ones until an AUC value of 1 is reached. Then, a 10-fold cross-validation is carried out to select the most complex model which still improves the AUC compared to the previous one. Decision trees are also used in *BADTrees* to represent interaction among SNPs and a method called bagging is introduced to increase the signal-to-noise ratio of the interacting SNPs. Bagging consists in bootstrapping a number of datasets from the original one, constructing a tree in each of the sets and finding similarities among them. In *BADTrees*, the most frequent SNPs among the trees are reported as associated with the phenotype.

StepPLR uses a penalized logistic regression model to quantify the association between the selected SNPs and the phenotype. It is an iterative algorithm where, based on a cost-complexity statistic which integrates either the Akaike Information Criterion or the BIC, SNPs or combination of SNPs are added or removed from the model in a series of forward selection and backward deletion steps. The model with the minimum cost is selected and the SNPs or combinations of SNPs included in the model are reported. Lastly, *SNPRuler* uses a rule-based classification model which introduces the

concept of rule utility and its derived upper bound to identify whether a rule can be further improved to increase its classification accuracy or not. *SNPRuler* begins by building a search tree to guide the search of interactions, where nodes represent SNPs and edges represent interactions between SNPs. The tree is built avoiding unnecessary expansions of child nodes, i.e. those whose utility's upper bound is lower than a certain threshold or its parent's utility. After the search tree is built, *SNPRuler* finds a number of top-ranked interactions (paths from the root to the leaf nodes) sorted by its utility measurement, calculates their p-value using the χ^2 statistic and writes the list to an output file.

3.1.4. Swarm Intelligent Methods

Swarm intelligence is a group of methods that falls under the category of metaheuristics. Metaheuristics are high level heuristic methods for exploring the search space, applicable to domains where the computational power of the information systems is insufficient, or the domain information is limited [77]. Swarm intelligence, as many of the metaheuristics, are nature-inspired methods that rely on the problem-solving ability that emerges from the interactions of simple information-processing units, or agents [78]. These are multiagent, decentralized and self-organized systems where the individual agents that integrate the system follow a rule-set that determines their behavior.

ACO is the most explored metaheuristic in epistasis detection. It relies on artificial ants (independent decision-making agents) to iteratively explore the SNP combination space. Pheromones are an implicit communication mechanism that ants use to guide the search. Whenever an ant explores a combination, it deposits a certain number of pheromones proportional to the association strength between the phenotype and the specific combination. Pheromones also evaporate over time, progressively reducing its effect. A probability function is used to decide which combination an ant should explore next based on the pheromone levels present on the combinations. The probability function also considers selecting a random combination under specified odds to avoid being trapped in local optima. After a fixed number of iterations are completed, the algorithm ends, and the result is a list of the most promising combinations visited by the ants. *MACOED* [32], *IACO* [67], *epiACO* [63] and *HiSeeker* [66] implement this

method faithfully, only exchanging the association measure and how the results are treated. *MACOED* uses the Pareto Optimal Set to select a group of candidate combinations from all explored and then applies a Pearson's χ^2 test to quantify its association. *IACO* and *epiACO* use the ratio between the MI and the Bayesian Network, and the ratio between the MI and the K2 score, respectively, to measure association. Both methods then proceed to calculate an inflection point on the association value to separate significant from irrelevant combinations. *HiSeeker*, as explained in Section 3.1.2, runs the ACO algorithm on a filtered group of SNPs. It uses Pearson's χ^2 test to evaluate the association, and the top-ranked combinations reported by the ACO algorithm are evaluated using the χ^2 test again to provide a Bonferroni-corrected p-value metric.

AntMiner [56] and *EACO* [61] innovate over the generic ACO algorithm by incorporating a heuristic into the probability function. *AntMiner* includes the addition of the Symmetrical Uncertainty and Spatially Uniform ReliefF onto the probability function, and segregates the ants into sub-colonies each exploring combinations of different sizes. It uses Pearson's χ^2 test as the association measurement. All explored combinations that surpass a certain χ^2 threshold are kept in what they call a Candidate Set, which is post-processed at the end to reduce false positives. *EACO*, on the other hand, uses the Multiple Threshold Spatially Uniform ReliefF as the heuristic of choice, and uses the ratio between MI and Gini index to assess association. Similarly to *IACO* and *epiACO*, significant combinations are identified by calculating an inflection point on the association metric.

CINOEDV [60] and *NHSA-DHSC* [73] use different swarm intelligence methods from the extensively seen ACO. *CINOEDV* implements the particle swarm optimization algorithm, where agents consist of particles with a defined position and velocity. The position represents the selected SNP combination, and from each position, its fitness or degree of association with the phenotype can be obtained using three different metrics: Co-Information, Normalized Co-Information and Contribution Co-Information. The velocity of each particle determines the next position to be explored. It depends on the current velocity, the best position found by the current particle and the best global position found by all particles. The algorithm initializes all particles' positions and velocities randomly and iterates for a fixed number of steps, storing the best position found on each iteration. It returns the list of positions sorted by the selected metric. The *NHSA-DHSC* method consists of two stages, a searching step that

implements the Niche Harmony Search Algorithm combining a harmony search algorithm with a niching technique, and a second stage where all found candidates are evaluated. Harmony search is a music-inspired swarm intelligent algorithm that mimics the improvisation process used by skilled musicians, where harmonies representing SNP combinations are iteratively explored following an improvisation process and the best harmonies are kept in a harmony memory [79]. The improvisation of new harmonies consists in choosing between pitch-adjusting previous harmonies and randomly exploring new ones. When the algorithm is stuck in a local optimum the niching algorithm is triggered, and the centroid and radius of the optimum point are included in a taboo table to be avoided by all future solutions, forcing the search algorithm to explore new areas in the solution space. *NHSA-DHSC* uses three different association metrics, kept in separate harmony memories, which are the K2-score, the Gini index and the joint entropy. After the *NHSA* algorithm ends, the three memories are joined into a common candidate pool and a G-test is performed on the resulting combinations to check for association with the phenotype.

3.1.5. Genetic Algorithms

Genetic Algorithms (GAs) are another group of metaheuristic methods which mimic the biological evolution process. GAs begin with a population of random solutions to a problem, encoded as chromosome-like data structures. The algorithm explores the solution space by evolving the current population into successive generations following a reproductive function. Reproduction consists of evaluation, selection, recombination and mutation steps. Solutions are evaluated using a fitness function, and reproductive opportunities are given proportionally to each individual according to its fitness. Selected individuals create offspring in a recombination operation, in which the two encoded solutions create two new offspring by selecting a (random) recombination point and swapping the subsequent fragments. Finally, a mutation step modifies some bits of the offspring following a specific probability function. The method evolves the population until a certain fitness of the solutions is achieved or the number of generations reaches the limit [80]. *GALE* [65] and *ATHENA* [31] use GAs to detect epistatic interactions.

GALE creates a rule-based classification system using a GA to generate a rule set.

The solutions of the population are ordered rule sets from which a rule-based classifier can be built. The fitness of a solution is measured as the average accuracy of its classifier in a k-fold cross-validation partition. *GALE* introduces the concept of spatial awareness to GAs by representing the population of solutions in a 2D grid and modifying the reproductive selection to take into account the proximity between solutions in the grid [81]. The final rule set obtained at the end of the GA is the solution provided by *GALE*.

ATHENA introduces Grammatical Evolution Neural Networks to the epistasis detection problem. Grammatical Evolution is a GA dedicated to the construction of computer programs, adapting the representation of solutions and the reproductive methods for this purpose. Solutions are variable length binary strings made of groups of 8 bits named codons, each encoding an integer. Codons are translated into rules following a predefined grammar specified in Backus-Naur form, and the translation of a complete solution is a program which can be evaluated using a fitness function [82]. *ATHENA* uses the coefficient of determination, R^2, as the fitness function to evaluate the different solutions considered. These solutions are made up of the SNPs used as input variables to the neural network, the network architecture itself and the weights associated to each of the connections. Using the BNF grammar, the different components of the solutions can be translated into a fully functional neural network. *ATHENA* also replaces the single-point crossover method from GAs with the Tree-Based Crossover method, which swaps a complete branch of the neural network to create offspring in order to avoid the uncertainty of recombining the network in its binary representation. *ATHENA* applies a 5-fold cross-validation to construct five different classification models and selects the model whose SNPs appear more consistently as the best model.

3.1.6. Random-Search-Based Methods

Lastly, a group of methods based on the random search algorithm can be identified. Random search stochastically samples the solution space for a number of iterations, evaluates each solution using a fitness function and saves the result with the best fitness value out of all the explored. *SNPHarvester* [74], *BEAM3* [58] and *BHIT* [59] are epistasis detection methods that belong to this group.

SNPHarvester implements an algorithm named *PathSeeker* to explore multiple combinations by the means of different local search iterations at random points of the combination space. *PathSeeker* follows a swapping technique, testing for all SNPs if any replacement in the combination can improve the χ^2 association value until no more replacements can be made. Once a predefined number of candidates has been found, a post-processing step is carried out to filter out spurious interactions by fitting a L_2 penalized logistic regression and reporting those interactions selected by the regression model.

BEAM3 uses a joint probability model between the SNP collection X, the interacting SNPs X_1 and a disease graph G; and the phenotype Y to determine the association present in the data. G is an undirected graph where nodes represent non-overlapping groups of SNPs from X_1 and edges represent interactions between groups. *BEAM3* explores the search space using Markov chain Monte Carlo sampling to update the selected SNPs in X_1 and its graph representation in G repeatedly. The sampling process adds or removes SNPs in or out of X_1 and updates the nodes and edges of G accordingly. After a number of iterations are completed, the algorithm ends and the best model is returned.

BHIT also resorts to a probability model to assess the association between genotypes and a phenotype, but this tool divides the genotype markers into different partitions. *BHIT* initializes the partition variable I by placing each SNP into a different partition and iteratively samples I using Markov chain Monte Carlo, maintaining the changes to I between iterations if the probability of the model increases. When the iterative process finishes, *BHIT* returns the different partitions in which the SNPs have been divided, the interacting SNPs being the ones grouped in the same partition as the phenotype variable.

3.2. Evaluation

This evaluation is separated into four parts: data simulation design, runtime evaluation, detection power analysis and false positive testing. In data simulation design, the pipeline created for simulating the datasets used in successive subsections is explained in detail. Runtime evaluation briefly compares how the different methods

perform in terms of execution time. Detection power measures the ability to locate combinations of SNPs associated with the phenotype under different simulation conditions. Lastly, false positive testing measures the ability to discern between significant and non-significant combinations.

Parameterization of the methods is consistent across the whole evaluation. In general terms, parameter selection was done either using the same values of the evaluation presented in its original work or following indications from the authors. The exception to this rule were swarm intelligent methods, where the number of iterations and agents is common to all methods in order to ensure a fair comparison. For most methods, there is a clear distinction in parameterization for third and fourth-order searches. When there is no interaction in the data, the parameters corresponding to the highest order admissible are selected. Appendix A covers, in detail, how the different parameters were chosen for each program.

3.2.1. Data Simulation Design

Many datasets were simulated for the evaluation of the methods, with varying features from one another in order to model different characteristics of the simulated population. The design goal of the simulation process was to generate a wide variety of datasets resembling real populations, therefore the parameterization used for modelling the population was chosen using estimates from real traits.

The simulation was carried out using *GAMETES* [83]. In this program, epistatic interactions are defined by penetrance tables, which describe the population frequencies of all possible allele sequences in a specific loci combination. The study considers model-driven interactions showing marginal effects, as well as model-free interactions with no marginal effects.

Penetrance tables with no marginal effects can be obtained natively through *GAMETES*, which follows a stochastic generation procedure to find epistatic relationships [83] under no model assumption. Model-driven penetrance tables, on the other hand, cannot be calculated within *GAMETES* and thus were obtained from *Toxo* [6], a MATLAB library developed during this thesis and explained in Chapter 6, which can compute penetrance tables from epistasis models. In this study, we employed the

widely used additive and threshold models proposed by Marchini et al. in [84], two models that define epistatic interactions with marginal effects.

Both *GAMETES* and *Toxo* calculate penetrance tables meeting a certain parameterization. The following list describes what these parameters are, and what criteria we used to select values:

- **Minor Allele Frequency (MAF)**. The frequency at which the second most common allele occurs in a given population. The distribution of observed susceptibility SNPs is skewed towards higher MAFs (>20 %) [85], and there is an increasing difficulty of detecting disease-causing variants with low MAF [86]. An accepted standard of MAF is 0.1, thus we have assayed values in the range [0.1, 0.4].
- **Heritability (h^2)**. The degree to which individual genetic variation accounts for the population phenotypic variation [87]. Heritability estimates of human traits for several medical conditions usually cluster in functional domains with its highest values between 70 and 80 % and the lowest ones between 2 and 30 % [88]. Therefore, we selected heritability values from the range [0.1, 0.8].
- **Prevalence ($P(D)$)**. The proportion of individuals from a population that carries a specific trait or suffers from a disease. Diseases can be categorized as rare if their prevalence is under 5×10^{-4} (fewer than 1 in 2000 people), and ultra-rare if it is under 2×10^{-5} (fewer than 1 in 50 000 people) [89]. For this simulation study, we have restricted prevalence values to be greater than 1×10^{-6}.

Table 3.2 lists all the parameters of the penetrance tables used throughout the evaluation. The criteria were to create penetrance tables of third and fourth-order, with MAF values of 0.10, 0.25 and 0.40 and heritabilities of 0.10, 0.25, 0.50 and 0.80 whose prevalence is above 1×10^{-6}. Model-driven tables cannot be obtained for every combination of MAFs, prevalence and heritability due to restrictions in the underlying mathematical model [6], resulting in a different number of tables according to the model. GAMETES, on the other hand, follows a probabilistic approach, which has problems to find model-free tables when increasing the interaction order, decreasing the MAF and increasing the heritability. Consequently, many combinations could not be obtained in a reasonable time.

Table 3.2: Interaction orders, MAFs, prevalence and heritability values of the penetrance tables used during the data simulation. Missing prevalence values correspond to a combination of parameters for which a penetrance table could not be obtained under the simulation conditions.

			ADDITIVE MODEL	THRESHOLD MODEL	NO MODEL (NME)
ORDER	MAF	H^2	P(D)	P(D)	P(D)
3	0.10	0.10	1.200×10^{-5}	6.460×10^{-2}	-
3	0.10	0.25	4.000×10^{-6}	2.556×10^{-2}	-
3	0.10	0.50	2.000×10^{-6}	1.327×10^{-2}	-
3	0.10	0.80	1.000×10^{-6}	8.417×10^{-3}	-
3	0.25	0.10	5.370×10^{-3}	4.775×10^{-1}	5.860×10^{-1}
3	0.25	0.25	1.153×10^{-3}	2.677×10^{-1}	4.923×10^{-1}
3	0.25	0.50	5.040×10^{-4}	1.545×10^{-1}	4.223×10^{-1}
3	0.25	0.80	3.060×10^{-4}	1.025×10^{-1}	-
3	0.40	0.10	2.546×10^{-1}	7.804×10^{-1}	5.163×10^{-1}
3	0.40	0.25	2.219×10^{-2}	5.870×10^{-1}	5.644×10^{-1}
3	0.40	0.50	8.545×10^{-3}	4.154×10^{-1}	5.019×10^{-1}
3	0.40	0.80	5.091×10^{-3}	3.075×10^{-1}	4.970×10^{-1}
4	0.10	0.10	-	1.256×10^{-2}	-
4	0.10	0.25	-	5.140×10^{-3}	-
4	0.10	0.50	-	2.590×10^{-3}	-
4	0.10	0.80	-	1.623×10^{-3}	-
4	0.25	0.10	2.340×10^{-4}	2.755×10^{-1}	4.201×10^{-1}
4	0.25	0.25	6.800×10^{-5}	1.320×10^{-1}	5.910×10^{-1}
4	0.25	0.50	3.100×10^{-5}	7.068×10^{-2}	-
4	0.25	0.80	1.900×10^{-5}	4.182×10^{-2}	-
4	0.40	0.10	3.628×10^{-2}	6.684×10^{-1}	4.720×10^{-1}
4	0.40	0.25	3.383×10^{-3}	4.464×10^{-1}	4.356×10^{-1}
4	0.40	0.50	1.374×10^{-3}	2.873×10^{-1}	-
4	0.40	0.80	8.220×10^{-4}	2.013×10^{-1}	-

From each penetrance table, 100 datasets were generated containing 500 SNPs from 2000 individuals (1000 cases and 1000 controls). Non-interacting loci were simulated using a MAF randomly sampled from the interval $[0.05, 0.5]$. In total, the data collection consists of 55 different epistatic relationships, 5500 datasets, 2.75 million SNPs and 11 million individuals.

Lastly, for the false positive testing, we also simulated 100 datasets with 500 SNPs from 2000 individuals (1000 cases and 1000 controls) containing no interaction. Loci for these datasets were also sampled from the MAF interval $[0.05, 0.5]$.

All the simulation configurations, epistasis models, penetrance tables and datasets

are publicly available at GitHub[1].

3.2.2. Runtime Evaluation

The runtime for each of the method's implementation was measured and compared using a single core of an Intel E5-2660. *SingleMI* is the only exception, since it uses NVIDIA GPUs, and thus it was run on an NVIDIA K20m. Fig. 3.1 compares the average runtime of all the studied tools for third and fourth-order analysis, across five repetitions. The first datasets of the additive model using MAF = 0.25 and $h^2 = 0.25$, both for third and fourth-order, were arbitrarily chosen for this purpose.

MDR, *EpiMiner* and *CINOEDV*'s fourth-order runtimes could not be measured as the maximum allocatable time in the evaluation system was equal to three days. *HiSeeker*'s runtime for fourth-order searches could not be measured as well, due to errors in the program which are not present during third-order searches.

Runtimes show a clear distinction between exhaustive and non-exhaustive methods: exhaustive methods are largely influenced by the interaction order, while non-exhaustive methods generally remain unaffected when moving from third to fourth-order. The only exceptions are *EpiMiner* and *CINOEDV*, programs which already show an extraordinarily large runtime despite using a dataset of moderate size, a runtime that is dependent on the combination size used during the search.

3.2.3. Detection Power

Using the simulated data, the detection power of the different methods can be measured as the ratio of datasets for which the epistatic interaction is correctly identified. Two different detection power metrics were used in the evaluation: the detection power considering all interactions reported by each method, and the detection power when only the first interaction reported is considered. Some implementations provide its output as a list of combinations in no particular order, therefore only the detection power of all reported interactions is obtainable. These methods include *BADTrees*, *StepPLR*, *MACOED*, *NHSA-DHSC*, *ATHENA* and *BHIT*. On the other side, some meth-

[1] https://github.com/UDC-GAC/epistasis-simulation-data

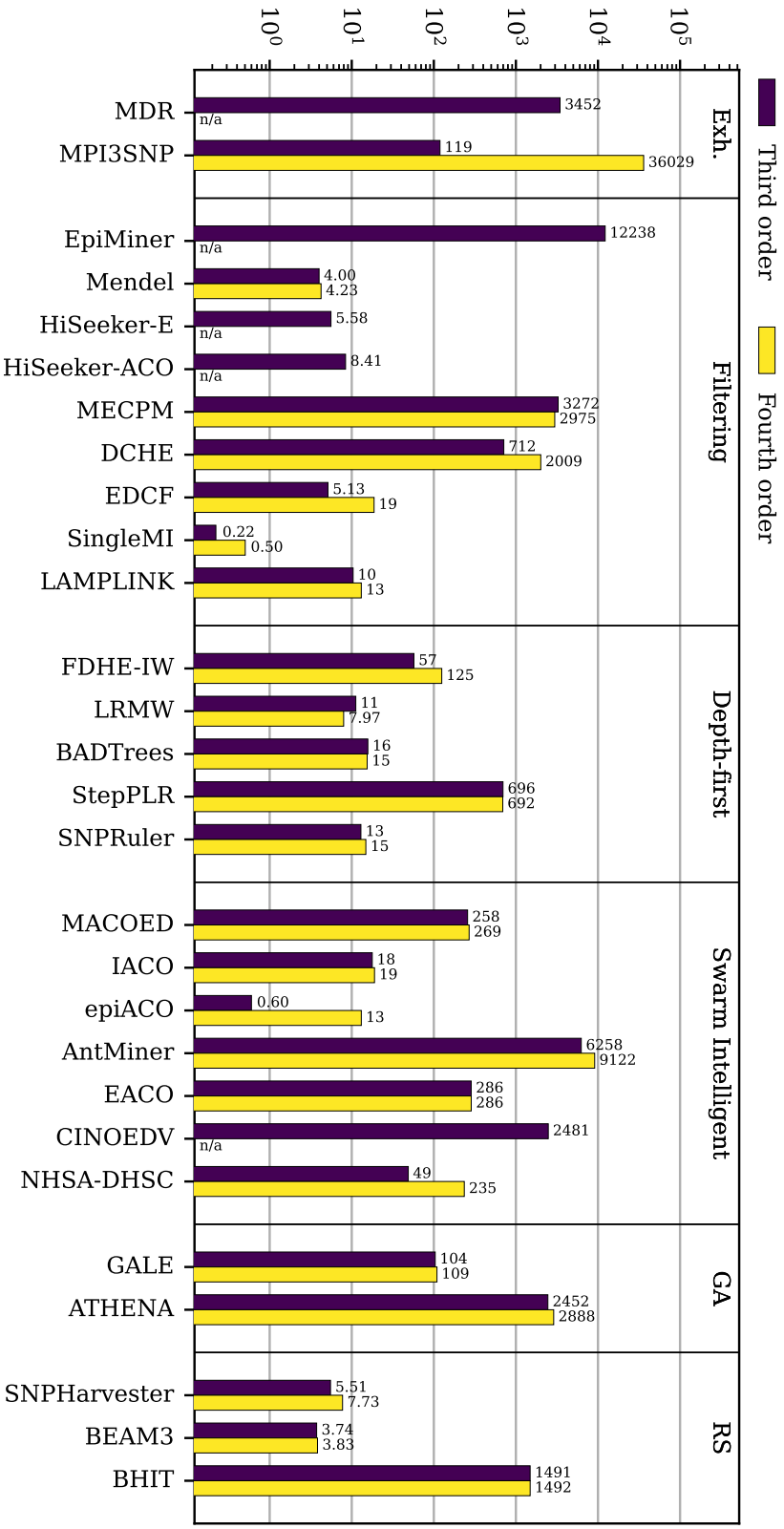


Figure 3.1: Average runtime, in seconds, of the different methods for third and fourth order interactions, using a dataset containing 500 SNPs, with 2000 samples per SNP. The experiments were run on an Intel E5-2660 except for *SingleMI*, which was run on an NVIDIA K20m.

ods only report a single interaction, thus both detection powers will be identical. These methods are *MDR*, *LRWM*, *GALE* and *BEAM3*. Additionally, *MDR*, *EpiMiner*, *CINOEDV* and *HiSeeker* could not be tested for fourth-order epistasis for the same reasons presented in the previous section.

Given the large number of configurations used, it is impractical to present all the individual results. Therefore, in this evaluation, the results are grouped by the interaction order and by the type of epistatic relationships, since these two account for most of the variation between results from the same method. The complete results are available in the supplementary material of [3].

Epistasis with Marginal Effects Following an Additive Model

Figs. 3.2 and 3.3 show the detection power of all methods when the data contains epistatic interactions displaying marginal effects under the additive interaction model. Fig. 3.2 represents the detection power from each method when all the reported interactions are considered, and Fig. 3.3 represents the same detection power when only the first reported interaction is considered.

Exhaustive methods reliably find the epistatic interaction in virtually all cases, and the correct interaction is the one always reported first. Conversely, genetic algorithms almost always miss the epistatic interaction. The remainder of the methods show mixed results and have to be discussed individually.

Out of the filtering methods, *EDCF* and *SingleMI* perform best with maximum detection powers even when considering only the first reported interaction. *MECPM* follows closely, although its detection power takes a toll when increasing the interaction order or when only the first reported interaction is considered. *LAMPLINK* and *EpiMiner* only perform well for third-order interactions when all the reported interactions are considered, *DCHE* shows mediocre results, and *Mendel* and *Hiseeker* cannot locate interactions whatsoever.

Depth-first methods show polarizing results. On the one hand, *FDHE-IW* perfectly identifies the correct interaction. *BADTrees* also shows a good detection power, although its output includes noise SNPs that do not contribute to the phenotypic outcome. *LRMW*, *StepPLR* and *SNPRuler*, on the other hand, obtain very low (if not zero)

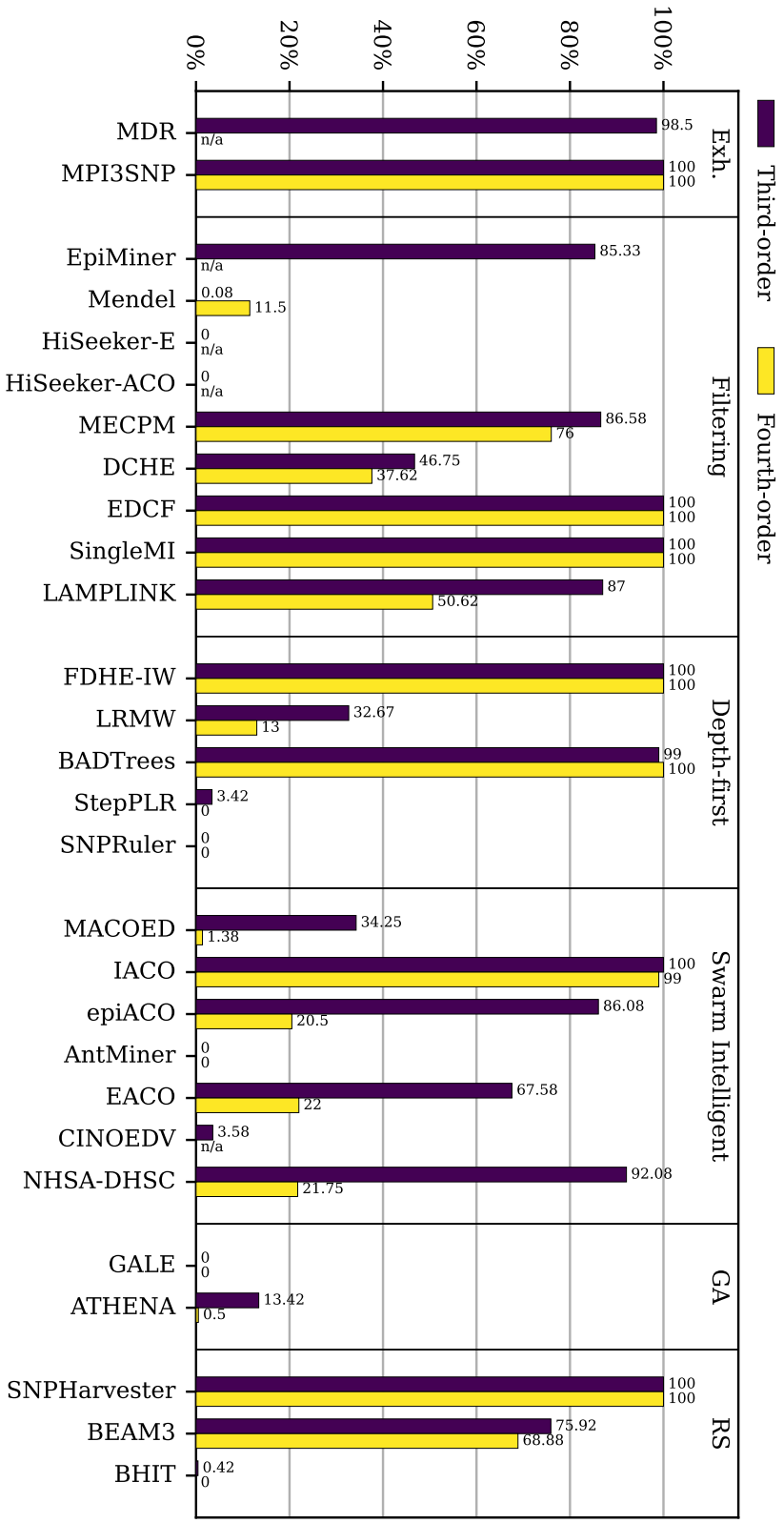


Figure 3.2: Average detection power of all methods when all reported interactions are considered, for the datasets containing epistasis under the additive model. Results not available are labeled accordingly.

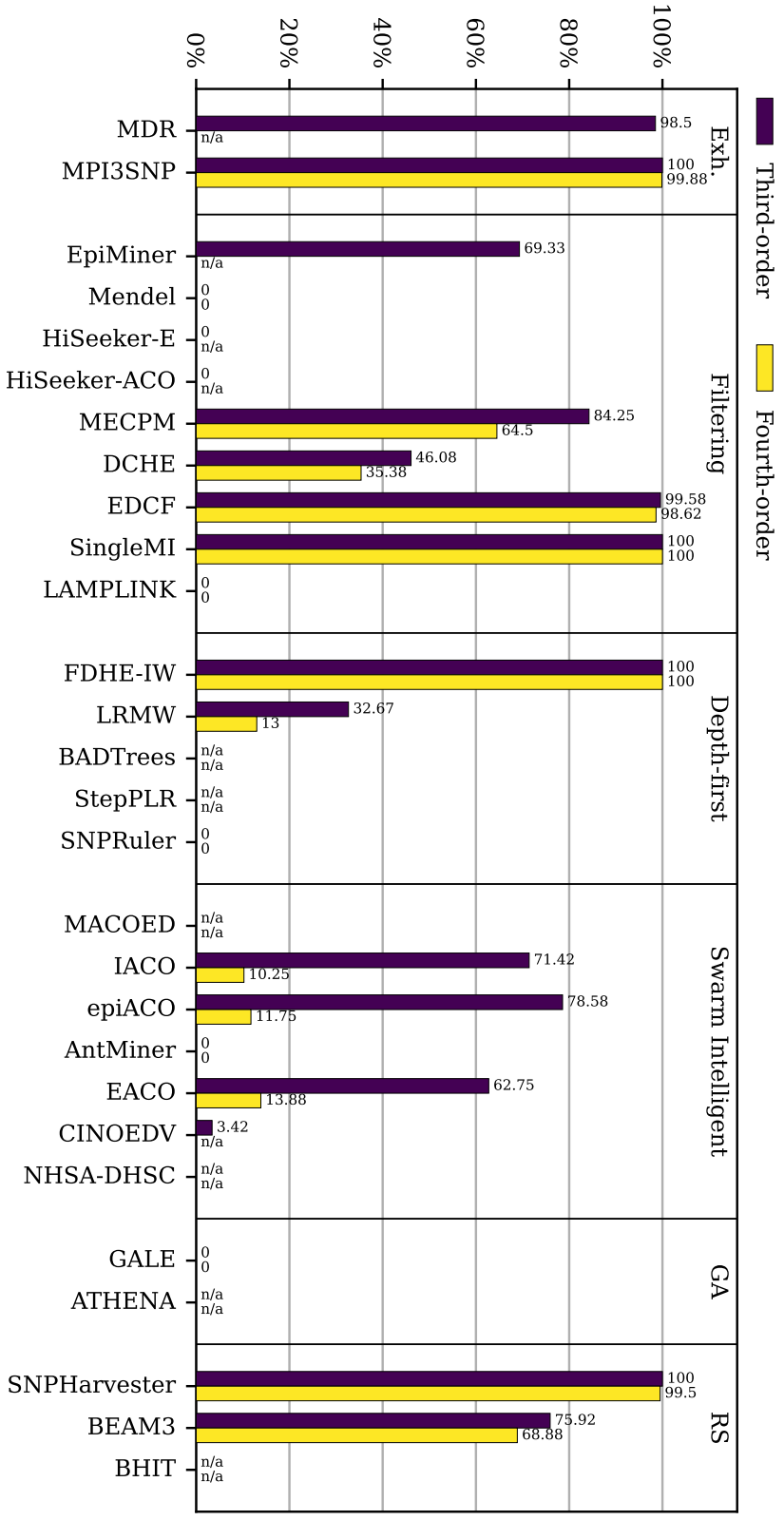


Figure 3.3: Average detection power of all methods when only the first reported interaction is considered, for the datasets containing epistasis under the additive model. Results not available are labeled accordingly.

detection powers.

Swarm intelligent methods show quite different results attending to the order of the interaction, with the only exception of *IACO*. This is coherent with the parameterization employed, since the number of iterations and agents (which control how much of the search space is explored) is kept constant throughout the evaluation despite the search space growing when the interaction order is increased. Swarm intelligent methods are also the most affected ones when only the first interaction is considered. *IACO* obtains almost perfect detection powers when all reported interactions are considered, however its detection power significantly drops when only the first one is used. *epiACO* and *NHSA-DHSC* also obtain high detection powers for third-order interactions, but their performance drops significantly when moving to fourth-order. *EACO* obtains mediocre results for third order, which also drop for fourth-order, and *MACOED*, *AntMiner* and *CINOEDV* obtain poor results.

Lastly, random-search based methods also obtain mixed results. *SNPHarvester* reports the correct interaction as the first one in almost all datasets. *BEAM3* obtains relatively good results, and *BHIT* is not capable of finding interactions.

Epistasis with Marginal Effects Following a Threshold Model

Figs. 3.4 and 3.5 show the detection power of all methods when the data contains epistatic interactions displaying marginal effects under the threshold interaction model. They represent the detection power when all interactions or only the first reported are considered, respectively.

Results for the threshold epistatic model are remarkably similar to those of the additive one, with some minor differences. Exhaustive methods noticeably drop their detection power, while genetic algorithms again fail to find any epistatic interaction.

Out of the filtering methods, *HiSeeker*, *DCHE* and *LAMPLINK* present the most drastic changes. *HiSeeker* goes from not being able to detect interactions at all under the additive epistatic model to reporting the correct interaction as the first one in almost all cases, and *DCHE* approximately doubles its previous detection power. *LAMPLINK*, on the contrary, drops its detection power down to zero. *EpiMiner* and *EDCF* slightly drop their detection powers. *SingleMI* and *Mendel* obtain very similar results

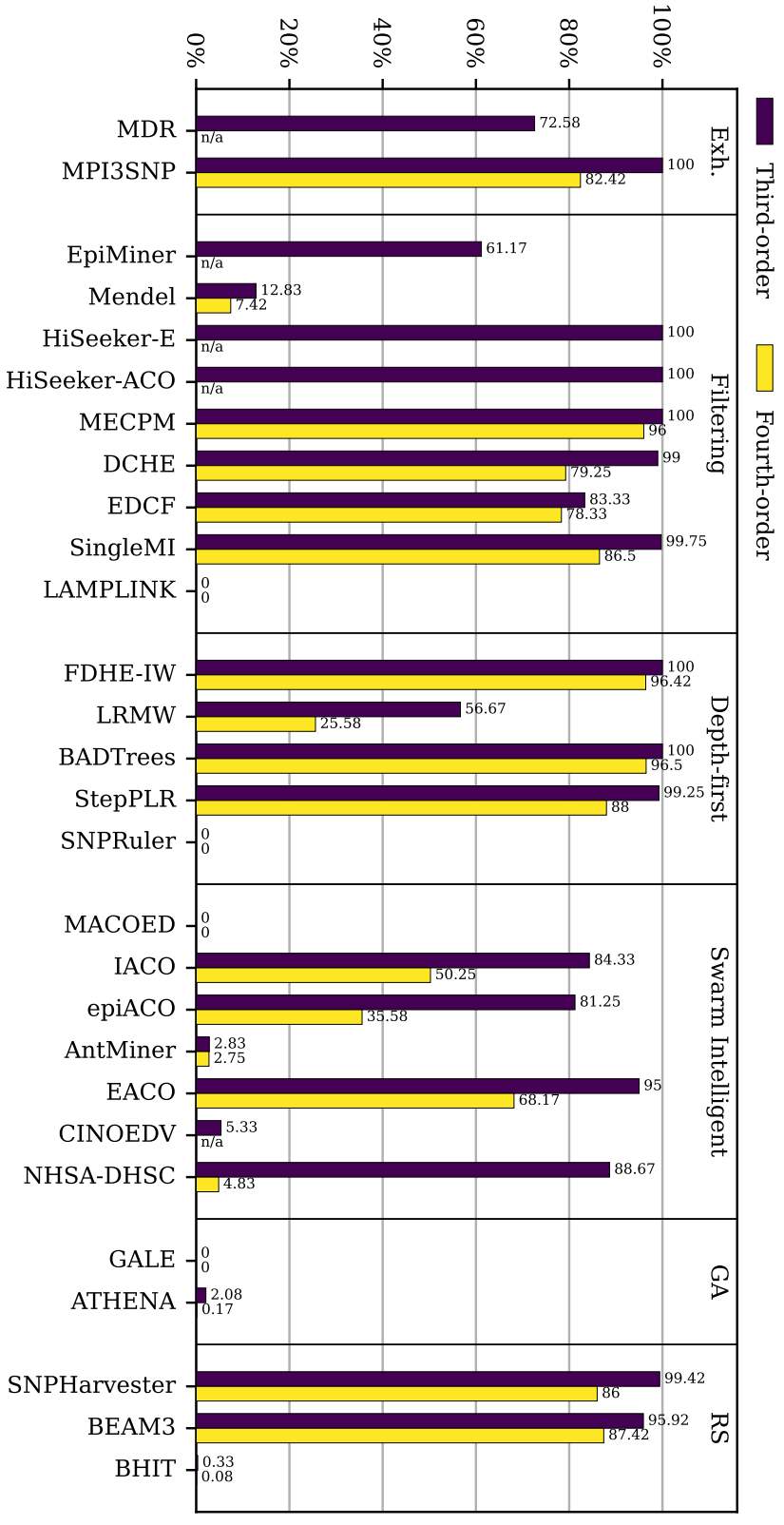


Figure 3.4: Average detection power of all methods when all reported interactions are considered, for the datasets containing epistasis under the threshold model. Results not available are labeled accordingly.

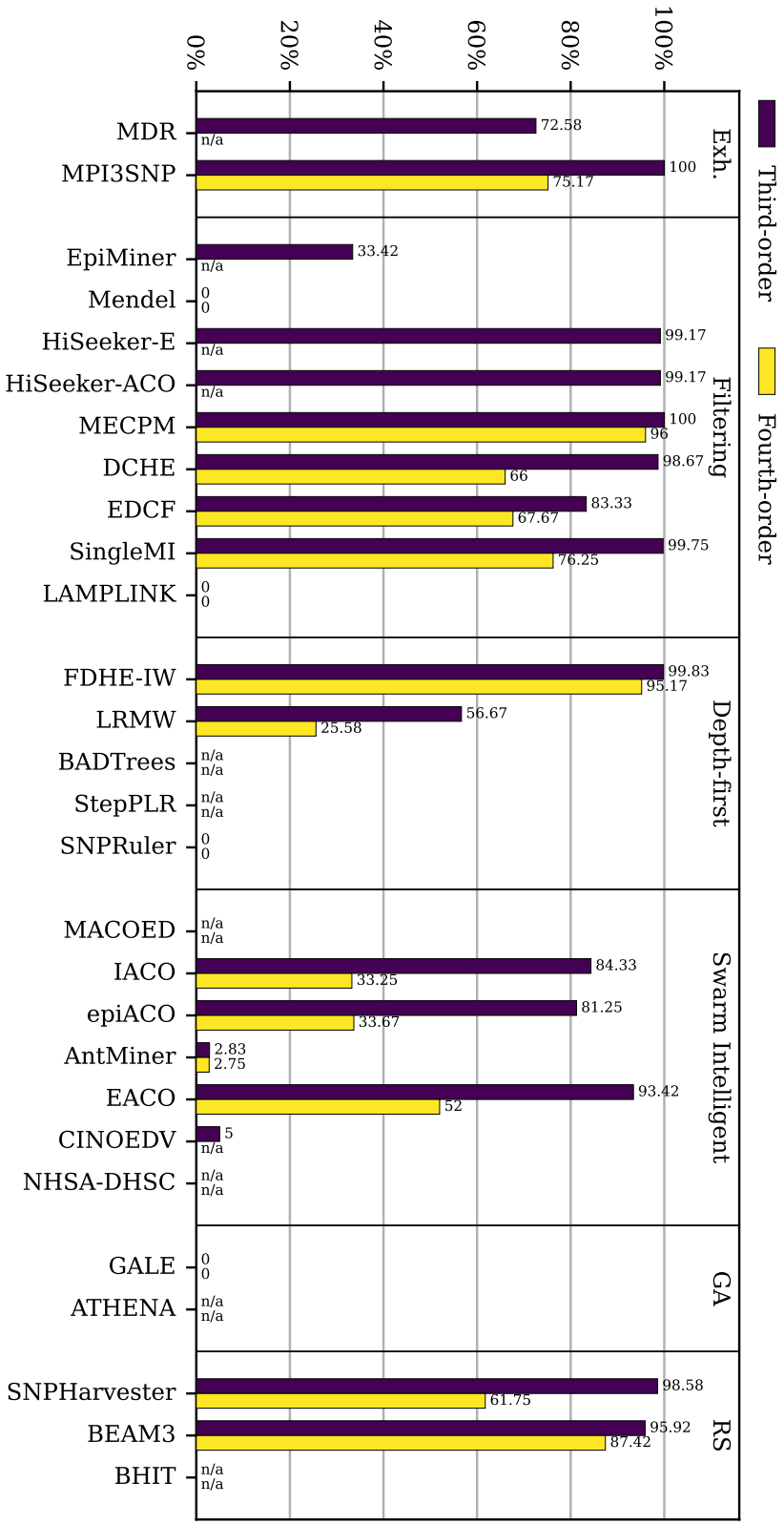


Figure 3.5: Average detection power of all methods when only the first reported interaction is considered, for the datasets containing epistasis under the threshold model. Results not available are labeled accordingly.

compared to previous additive model results, the former with high powers and the latter with powers next to zero.

Depth-first methods obtain similar results compared to their previous values, with the only exception of *StepPLR*. *FDHE-IW* and *BADTrees* obtain almost the same detection powers as with the additive model, while LRMW slightly improves it. *StepPLR*, on the contrary, increases its detection power from next to 0 % to almost 100 %.

Swarm intelligent algorithms show slight variations from their previous detection powers, with *epiACO*, *AntMiner*, *CINOEDV* and *NHSA-DHSC* showing similar results while *EACO* significantly increasing its detection power and *IACO* and *MACOED* showing a noticeable decrease.

Random-search based algorithms also show minor variations compared to the results with the additive model. *SNPHarvester* noticeably drops its detection power for fourth-order interactions, both when all and only the first reported interactions are considered, while maintaining its third-order power. *BEAM3*, on the opposite, increases its detection power, and *BHIT* remains near zero.

Epistasis with No Marginal Effects under No Interaction Model

Figs. 3.6 and 3.7 show the detection power of all methods when the data contains epistatic interactions displaying no marginal effects under no interaction model.

Detection powers when no marginal effects are present show a completely different story than the previous two interaction models. Out of all the methods tested, only exhaustive approaches are capable of consistently locating interactions that show no marginal effects. The only other methods that show a detection power above zero for third-order interactions are *DCHE*, *EDCF* and *SNPRuler*. *DCHE* and *EDCF* show a detection power much lower than in scenarios with marginal effects. *SNPRuler*, however, was unable to find any interaction in previous interaction models and now it is one of the three methods capable of finding the interaction in a fraction of all datasets.

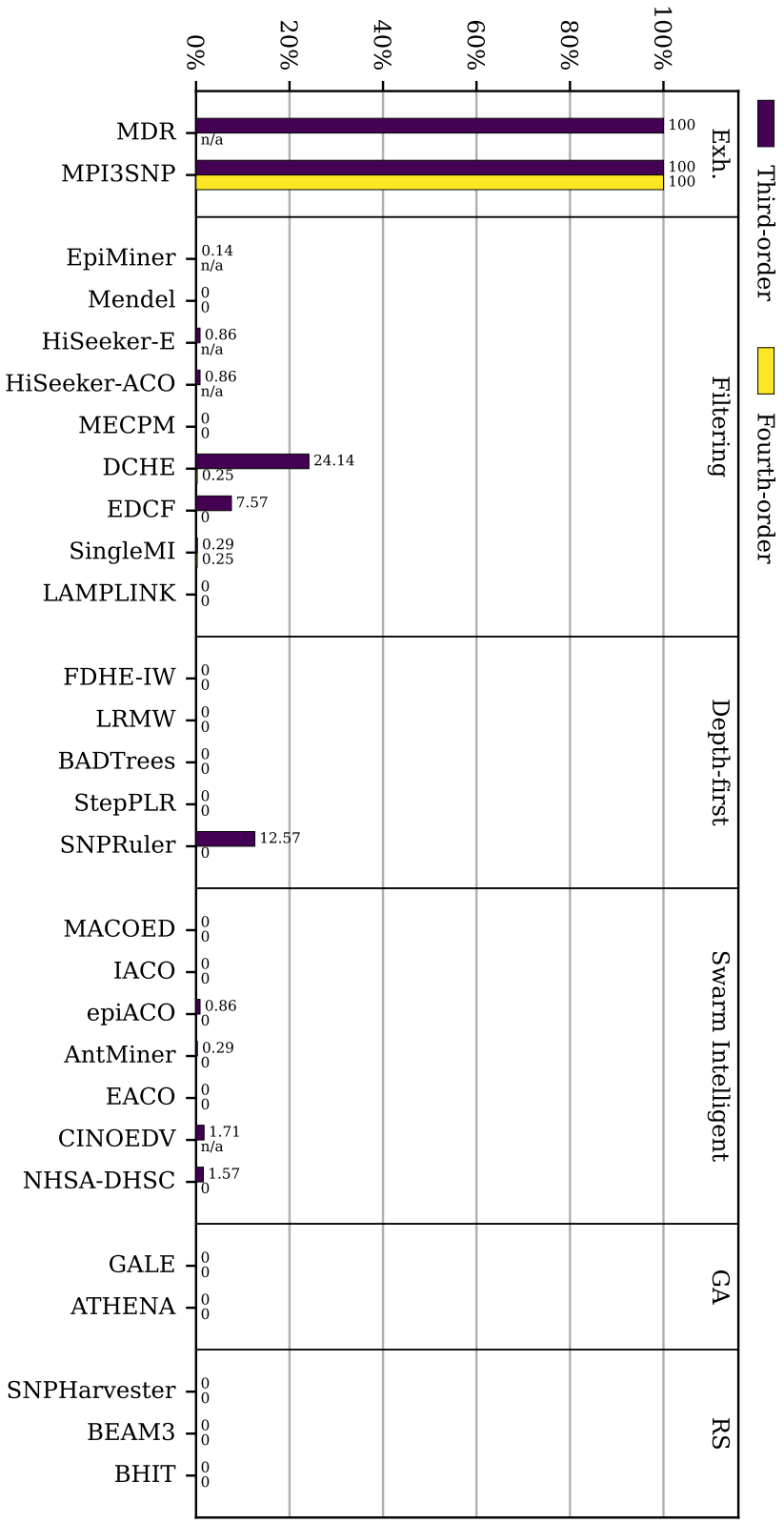


Figure 3.6: Average detection power of all methods when all reported interactions are considered, for the datasets containing epistasis under no particular model and showing no marginal effects. Results not available are labeled accordingly.

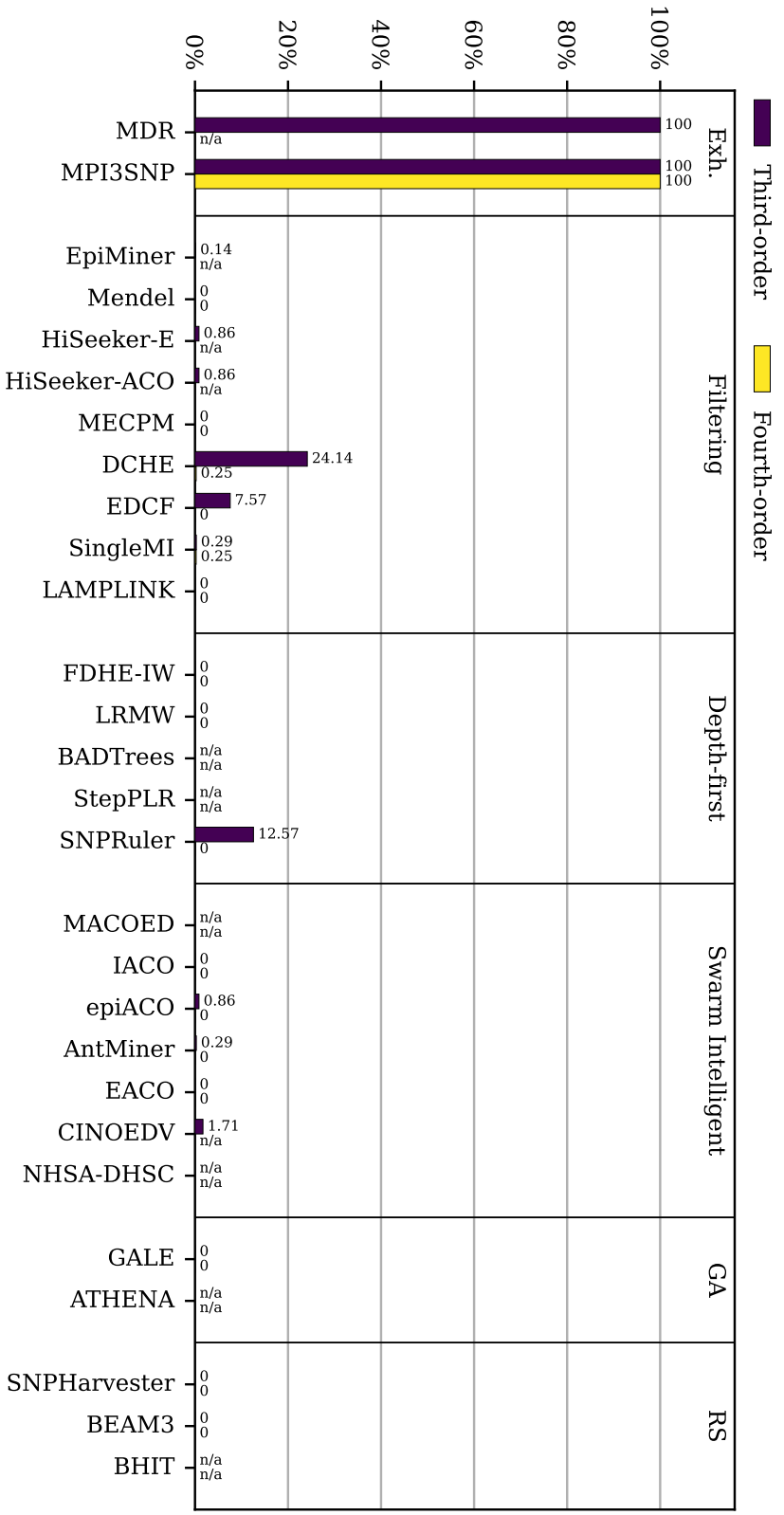


Figure 3.7: Average detection power of all methods when only the first reported interaction is considered, for the datasets containing epistasis under no particular model and showing no marginal effects. Results not available are labeled accordingly.

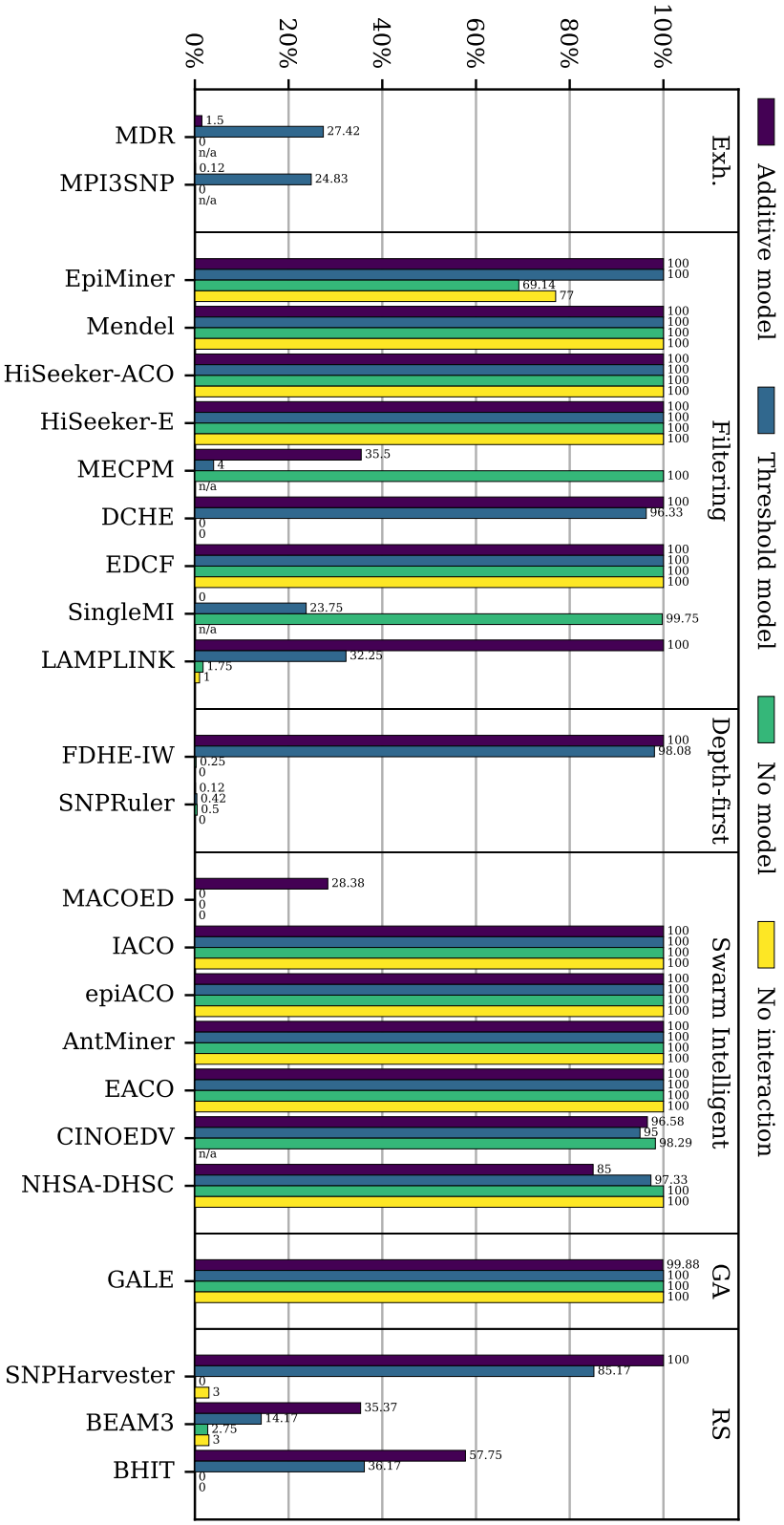


Figure 3.8: FWER of all applicable methods, using datasets containing interactions showing marginal effects and following an additive and threshold model, containing interactions showing no marginal effects and under no epistasis model, and containing no interactions.

3.2.4. False Positive Testing

False positive testing evaluates whether or not non-interacting loci are reported when searching for epistasis. To measure false positive detection the FWER was used, defined as the ratio of datasets where any combination of non-interacting SNPs is reported.

FWER was measured using the previously presented datasets that contain epistatic interactions showing marginal effects following additive and threshold models, as well as those showing no marginal effects under no model assumption. Additionally, FWER was also measured on datasets containing no epistatic interactions.

FWER could not be measured for all epistasis detection methods and for all scenarios presented. Implementations that are forced to return any number of unordered SNP combinations could not be included in this evaluation. This includes *LRMW*, *BADTrees*, *StepPLR* and *ATHENA*. The FWER for programs that return a fixed number of ordered combinations was measured considering only the first reported interaction. In this scenario, the FWER is the complementary measure of the detection power when only the first reported interaction is considered, and cannot be measured when there is no epistasis. This includes *MDR*, *MPI3SNP*, *MECPM*, *SingleMI* and *CINOEDV*.

Fig. 3.8 represents the FWER for the methods evaluated. The figure shows that false positives have a significant presence in most of the methods. These results can be divided into three categories: methods that report many false positives regardless of the data, methods that report few false positives and methods that show very different results depending on the epistasis model or presence/absence of epistasis.

Most of the methods fall under the first category. *EpiMiner*, *Mendel*, *HiSeeker*, *EDCF*, *IACO*, *epiACO*, *AntMiner*, *EACO*, *CINOEDV*, *NHSA-DHSC* and *GALE* almost always include false positives in its output. On the opposite, *MDR*, *MPI3SNP*, *SNPRuler*, *MACOED* and *BEAM3* keep their FWER under control.

As for methods showing different results depending on the dataset, the most common behaviour is to report false positives on the presence of marginal effects. *DCHE*, *LAMPLINK*, *SNPHarvester*, *BEAM3* and *BHIT* report almost no false positives when there are no marginal effects or there is no interaction. On the other hand, *MECPM* and *SingleMI* show an erratic behavior of the FWER for different datasets.

3.3. Discussion

It is clear from the previous detection power results that current epistasis detection methods, outside of the exhaustive approach, rely on the existence of marginal effects to locate the epistatic interaction. The best non-exhaustive approach for interactions showing no marginal effects is *DCHE*, with a detection power of 24.14 % for third-order interactions which completely disappears when the order is increased.

Table 3.3 summarizes the results for epistatic interactions with marginal effects. For each program the average detection power is calculated, differentiating between third and fourth-order. *FDHE-IW*, *MPI3SNP*, *SingleMI*, *SNPHarvester*, *BADTrees*, *MECPM*, *EDCF* and *BEAM3* show average detection powers above 80 %, both for third and fourth-order epistasis search. *IACO*, *NHSA-DHSC*, *epiACO* and *EACO* despite showing detection powers above 80 % for third-order searches, immediately drop by more than 20 points when moving to fourth-order. *MDR*, on the other hand, cannot obtain fourth-order results in a reasonable runtime, and therefore its success is also limited to third-order.

Genetic algorithms are the only family of methods that is not represented on the upper half of the table. Swarm intelligent methods, despite their mediocre results for fourth-order searches, demonstrate good results for third-order, indicating that the number of agents and iterations selected has to take the order of the interactions into consideration. Genetic algorithms, on the other hand, do not find any success under any of the conditions presented.

Table 3.4 synthesizes the results for false positive testing, showing the average FWER while differentiating between the presence or absence of epistasis. The table shows that, when looking for epistasis, only five methods report false positives in less than 25 % of the datasets tested. These methods are *SNPRuler*, *MPI3SNP*, *MDR*, *BEAM3* and *MACOED*. Only three of these five methods show good detection powers, which questions if the good false positive results of *SNPRuler* and *MACOED* are linked to their lack of detection.

When epistasis is not present, eight methods can obtain FWER close to zero. Out of these eight, half of them obtain reasonably high detection powers when epistasis is present, including *DCHE*, *FDHE-IW*, *BEAM3* and *SNPHarvester*. The other half, com-

Table 3.3: Summary of the detection power results when marginal effects are present.

	THIRD-ORDER		FOURTH ORDER	
RANK	METHOD	POWER (%)	METHOD	POWER (%)
1	FDHE-IW	100.00	BADTrees	97.90
2	MPI3SNP	100.00	FDHE-IW	97.85
3	SingleMI	99.88	SingleMI	91.90
4	SNPHarvester	99.71	SNPHarvester	91.60
5	BADTrees	99.50	MPI3SNP	89.45
6	MECPM	93.29	MECPM	88.00
7	IACO	92.17	EDCF	87.00
8	EDCF	91.67	BEAM3	80.00
9	NHSA-DHSC	90.38	IACO	69.75
10	BEAM3	85.92	DCHE	62.60
11	MDR	85.54	StepPLR	52.80
12	epiACO	83.67	EACO	49.70
13	EACO	81.29	epiACO	29.55
14	EpiMiner	73.25	LRMW	20.55
15	DCHE	72.88	LAMPLINK	20.25
16	StepPLR	51.33	NHSA-DHSC	11.60
17	HiSeeker-ACO	50.00	Mendel	9.05
18	HiSeeker-E	50.00	AntMiner	1.65
19	LRMW	44.67	MACOED	0.55
20	LAMPLINK	43.50	ATHENA	0.30
21	MACOED	17.13	BHIT	0.05
22	ATHENA	7.75	GALE	0.00
23	Mendel	6.46	SNPRuler	0.00
24	CINOEDV	4.46	CINOEDV	-
25	AntMiner	1.42	EpiMiner	-
26	BHIT	0.38	HiSeeker-ACO	-
27	GALE	0.00	HiSeeker-E	-
28	SNPRuler	0.00	MDR	-

posed of *BHIT*, *MACOED*, *SNPRuler* and *LAMPLINK*, obtains poor detection powers which, again, questions if the good false positive results are linked to their weak detection ability.

Results also suggest a possible two-stage strategy for finding new epistatic interactions with marginal effects, in a reasonable execution time and with a low probability of including false positives: combining *FDHE-IW* with *MPI3SNP*. *FDHE-IW* could be used first to discern whether a dataset contains epistasis, due to its high detection power, low runtime and low FWER under the assumption of no epistasis. If any candidate combination is reported, *MPI3SNP* would then be used to analyze only the re-

Table 3.4: Summary of the FWER results.

	With epistasis		Without epistasis	
Rank	Method	FWER (%)	Method	FWER (%)
1	SNPRuler	0.29	BHIT	0.00
2	MPI3SNP	5.44	DCHE	0.00
3	MDR	11.19	FDHE-IW	0.00
4	BEAM3	16.11	MACOED	0.00
5	MACOED	18.84	SNPRuler	0.00
6	SingleMI	25.18	LAMPLINK	1.00
7	MECPM	29.47	BEAM3	3.00
8	BHIT	35.60	SNPHarvester	3.00
9	LAMPLINK	51.98	EpiMiner	77.00
10	SNPHarvester	76.62	AntMiner	100.00
11	DCHE	79.18	EACO	100.00
12	FDHE-IW	79.98	EDCF	100.00
13	NHSA-DHSC	87.42	epiACO	100.00
14	EpiMiner	93.03	GALE	100.00
15	CINOEDV	96.35	HiSeeker-ACO	100.00
16	GALE	99.98	HiSeeker-E	100.00
17	AntMiner	100.00	IACO	100.00
18	EACO	100.00	Mendel	100.00
19	EDCF	100.00	NHSA-DHSC	100.00
20	epiACO	100.00	CINOEDV	-
21	HiSeeker-ACO	100.00	MDR	-
22	HiSeeker-E	100.00	MECPM	-
23	IACO	100.00	MPI3SNP	-
24	Mendel	100.00	SingleMI	-

ported SNPs due to its high detection power and low FWER, under the assumption of epistasis, while circumventing the high runtime associated with exhaustive methods due to the previous filtering step.

To conclude the evaluation, it is worth mentioning that *BADTrees*, a method that achieves very good results in terms of detection power, does not implement any statistical method that allows the elimination of false positives, which detracts from the tool's applicability.

3.4. Concluding Remarks

This chapter provides an overview of the current methods dedicated to high-order epistasis detection, as well as a comparison of the results achieved by the different implementations in terms of detection power and type I error rate. For each method, its detection power and error rates were measured using more than 5000 synthetic datasets, each one involving different simulation conditions in order to make a fair comparison.

Results show that many of the current epistasis detection methods, regardless of the strategy used, can reliably find the epistatic interaction when marginal effects are present, although their detection power generally decreases with the order of the interaction. The only exception are genetic algorithms, as none of the two methods implementing this strategy can consistently find interactions. Non-exhaustive methods, however, behave very poorly when marginal effects are absent. In this scenario the only option that seems to reliably locate the interactions is the exhaustive strategy, with the subsequent exponential runtime complexity associated with the order of the interaction searched.

False positives' evaluation speaks of a different story. Out of the 27 methods compared, *BEAM3* is the only method capable of reliably finding epistasis while keeping type I errors to a minimum. Moving forward, authors should give more importance to type I error control. Methods that consistently report false positives lose much of their value, since their usability is restricted to the verification of previous findings. Looking for new epistatic interactions requires implementing a tight false positive control in order to avoid reporting false associations.

Even though exhaustive methods show one of the largest runtimes, and this discrepancy should be exacerbated for higher orders of interaction due to its exponential computational complexity, they are the only approach that locates epistasis in all scenarios. For this reason, this thesis continued to focus on the exhaustive method, trying to mitigate the growth of the runtime by exploiting clusters as efficiently as possible. Chapters 4 and 5 propose a vector and a distributed algorithm, respectively, in order to accelerate the search.

Chapter 4

SIMD Implementation of the Association Test

Most modern CPU architectures, if not all, include Vector Processing Units (VPUs) in their processing cores. Although compilers incorporate automatic vectorization techniques to exploit the VPUs, they show limitations on what can be automatically vectorized, and the performance obtained is not always the optimal. In this chapter, we propose an exhaustive epistasis detection algorithm to find interactions of any order that makes use of an explicit Single Instruction Multiple Data (SIMD) implementation to maximize the performance per core. Starting from the MI association test introduced in Chapter 1, Section 4.1 describes how the MI association test can be implemented in C++ and vectorized using AVX Intrinsics. Section 4.2 describes how the exhaustive search can be adapted to the MI vector algorithm. Section 4.3 presents the experimental evaluation, comparing the performance achieved by the explicit vector implementations using AVX Intrinsics with the performance achieved by the automatic vectorization that compilers offer and the performance of the MPI3SNP implementation. At last, Section 4.4 discusses the conclusions extracted from this study, reflects on its limitations and comments on the next steps to be followed.

4.1. Vector Implementation of the MI Association Test

This section covers the explicit vectorization of the MI association test introduced in Section 1.3, using 256-bit and 512-bit AVX Intrinsics from the *AVX2* and *AVX512BW* extensions. This section also addresses several optimizations to improve the performance of the vectorized codes.

The *AVX2* vector extension was first introduced with the Intel Haswell microarchitecture (2013) while the *AVX512BW* extension first appeared in the Skylake-X processors (2017) of the Skylake microarchitecture. These two vector extensions are used not only to optimize the runtime of the epistasis detection tool on a long list of CPUs, but also to compare the performance that the two vector widths offer.

4.1.1. Vectorization of the Genotype Table Calculation

Listing 4.1 shows a C++ implementation of the genotype table construction operation described in Section 1.3.1. The function, named `combine`, takes as arguments a genotype table representing the combination of any number of SNPs, a genotype table of a singular SNP and a genotype table where the results will be stored. Note that the template argument `uint64_t`, common to all genotype table classes, indicates the type used to store the binary information (Lines 11–12). Since the *x86_64* instruction set operates with 64-bit integers, this type is ideal to hold the genotype information, each value representing the information of 64 individuals, and each computation operating with 64 individuals at once. The function calls the `combine_subtable` subroutine twice to combine each of the two subtables for cases and controls. This function consists of three nested *for* loops, with the two outermost loops (Lines 4 and 5) combining the different rows of the input genotype tables. The innermost loop (Line 6) iterates over the different `uint64_t` values of the selected rows, reading one value from each input table, calculating its intersection and storing the result in the output table. The computational time complexity of this operation is $O(3^k m)$, where m is the number of individuals in the data and k is the size of the combination.

The function `combine_subtable` is the one implementing the computation of a genotype subtable from two previous subtables, and thus our target for vectorization. In this function, we can identify a vectorization opportunity at the innermost loop,

Listing 4.1: Genotype table combination function

```
1  inline void combine_subtable(const uint64_t *gt_tbl1, const size_t size1,
2    const uint64_t *gt_tbl2, const size_t words, uint64_t *gt_tbl3)
3  {
4    for (size_t i = 0; i < size1; i++) {
5      for (size_t j = 0; j < 3; j++) {
6        for (size_t k = 0; k < words; k++) {
7          gt_tbl3[(i * 3 + j) * words + k] = gt_tbl1[i * words + k] &
8                                             gt_tbl2[j * words + k];
9  }}}}
10
11 void combine(const GenotypeTable<uint64_t> &t1,
12   const GenotypeTable<uint64_t> &t2, GenotypeTable<uint64_t> &out)
13 {
14   combine_subtable(t1.cases, t1.size, t2.cases, t1.cases_words, out.cases);
15   combine_subtable(t1.ctrls, t1.size, t2.ctrls, t1.ctrls_words, out.ctrls);
16 }
```

where the intersection of two rows from two tables is calculated by performing as many bitwise *AND* operations as values contained in the row (Lines 7–8). This operation is already exploiting the data-parallelism that 64-bit operations offer, as the information of a singular individual is stored in a single bit of the data type `uint64_t`. With the introduction of 256 and 512-bit AVX instructions, the throughput of this operation can be multiplied.

Listing 4.2 shows the implementation using 256-bit AVX Intrinsics from *AVX2*. For simplicity, we assume that the number of bytes in a row of the genotype table is divisible by the vector unit width. This is achieved by padding the rows with zeros if the number of individuals is not divisible by the width of the vector unit, and it will not influence the result of the following *popcount* operation. The new figure replaces the C++ code corresponding to the two array accesses, the *AND* and the store operations with AVX loads, *ANDs* and store intrinsics (Lines 13–19). With just the introduction of the AVX Intrinsics, there is a front-end bound problem in which the CPU wastes many clock cycles waiting for instructions to be fetched. Therefore, to correct this behaviour, the middle loop was unrolled completely so that the three rows from the second genotype table are processed concurrently.

The 512-bit vector implementation using intrinsics from the *AVX512BW* extension is almost identical to the one shown in Listing 4.2, and thus it was omitted. The only differences are the name of the functions that implement the same operations for a 512-bit width, the types that these operations use and the step of the innermost loop,

Listing 4.2: Genotype table combination function vectorized with *AVX2* Intrinsics

```
1  inline void combine_subtable(const uint64_t *gt_tbl1, const size_t size1,
2    const uint64_t *gt_tbl2, const size_t words, uint64_t *gt_tbl3)
3  {
4    const __m256i *ptr1 = gt_tbl1;
5    for (size_t i = 0; i < size1; ++i) {
6      const __m256i *ptr2_1 = gt_tbl2 + 0 * words;
7      const __m256i *ptr2_2 = gt_tbl2 + 1 * words;
8      const __m256i *ptr2_3 = gt_tbl2 + 2 * words;
9      __m256i *ptr3_1 = gt_tbl3 + (i*3+0) * words;
10     __m256i *ptr3_2 = gt_tbl3 + (i*3+1) * words;
11     __m256i *ptr3_3 = gt_tbl3 + (i*3+2) * words;
12     for (size_t k = 0; k < words; k += 4) {
13       __m256i y0 = _mm256_load_si256(ptr1++);
14       __m256i y1 = _mm256_load_si256(ptr2_1++);
15       __m256i y2 = _mm256_load_si256(ptr2_2++);
16       __m256i y3 = _mm256_load_si256(ptr2_3++);
17       _mm256_store_si256(ptr3_1++, _mm256_and_si256(y0, y1));
18       _mm256_store_si256(ptr3_2++, _mm256_and_si256(y0, y2));
19       _mm256_store_si256(ptr3_3++, _mm256_and_si256(y0, y3));
20 }}}
```

which doubles the one used in the 256-bit implementation.

4.1.2. Vectorization of the Contingency Table Calculation

Listing 4.3 shows a C++ function implementing the contingency table construction. This function does not faithfully implement the operation described in Section 1.3.2. Instead, it combines the genotype table and contingency table calculations in a single step in order to avoid the numerous store operations that the genotype table construction requires. The function `combine_and_popcnt` consists of two calls to the subroutine `popcnt_subtable` to compute the contingency subtables of the two new genotype subtables, and this subroutine consists of the same three nested loops as Listing 4.1. However, instead of immediately storing the multiple `uint64_t` values resulting from the bitwise *AND* operations, the *popcount* operation is called, the results from the same row are summed up in a single `uint32_t` value and the sum is stored in the contingency table (Lines 9–10). In contrast with the genotype table class, the contingency table class uses the `uint32_t` type (passed as a template argument in Line 14) to represent the total count of individuals having a particular genotype because it can contain a large enough integer, and for the convenience of matching the size of a `float` value which will be useful later during the vectorization of the MI computation. The compu-

Listing 4.3: Contingency table calculation function

```
1  inline void popcnt_subtable(const uint64_t *gt_tbl1, const size_t size1,
2    const uint64_t *gt_tbl2, const size_t words, uint32_t *ct_tbl,
3    const size_t ct_size)
4  {
5    for (size_t i = 0; i < size1; i++) {
6      for (size_t j = 0; j < 3; j++) {
7        ct_tbl[i * 3 + j] = 0;
8        for (size_t k = 0; k < words; k++) {
9          ct_tbl[i * 3 + j] += std::bitset<64>(gt_tbl1[i * words + k] &
10                                              gt_tbl2[j * words + k]).count();
11 }}}}
12
13 void combine_and_popcnt(const GenotypeTable<uint64_t> &t1,
14   const GenotypeTable<uint64_t> &t2, ContingencyTable<uint32_t> &out)
15 {
16   popcnt_subtable(t1.cases, t1.size, t2.cases, t1.cases_words, out.cases, out.size);
17   popcnt_subtable(t1.ctrls, t1.size, t2.ctrls, t1.ctrls_words, out.ctrls, out.size);
18 }
```

tational time complexity of this operation is also $O(3^k m)$, with m being the number of individuals in the data and k the size of the combination.

The main difference between the codes for calculating genotype and contingency tables (Listings 4.1 and 4.3, respectively) is the presence of the *popcount* operation. Up until very recently, with the introduction of the Intel Ice Lake processors, there was no AVX vector instruction implementing a *popcount*. Muła et al., in [90], have already explored this problem and they proposed multiple algorithms for implementing population counts using the *AVX2* extension. Furthermore, in their GitHub repository[1], they have developed updated versions of the algorithms to make use of the more recent *AVX512BW* and *AVX512VBMI* extensions.

Deciding which algorithm runs the fastest is not trivial and cannot be measured in isolation, as interleaving additional loads and bitwise *AND* operations in between *popcounts* will undoubtedly affect the performance of the function as a whole. For this reason, we implemented multiple versions of the `combine_and_popcount` function and compared the performance of each choice. Table 4.1 includes the elapsed time during the computation of a contingency table for the different implementations of the function, running on an Intel 6240 processor, the one used for the experimental evaluation in Section 4.3, and compiled with GCC version 8.3. The table considers:

[1] https://github.com/WojciechMula/sse-popcount

Table 4.1: Elapsed time, in seconds, during the computation of a single contingency table using different operation widths and *popcount* implementations. The table highlights with green background the best times using the *AVX512BW* extension, and with red text the best times using the *AVX2* extension

AND WIDTH	POPCNT WIDTH	POPCNT ALGORITHM	INDIVIDUALS COUNT					
			256	512	1024	2048	4096	8192
512	512	harley seal	8.60×10^{-7}	8.60×10^{-7}	1.02×10^{-6}	1.31×10^{-6}	1.88×10^{-6}	1.60×10^{-6}
512	512	lookup	2.47×10^{-7}	2.47×10^{-7}	3.39×10^{-7}	5.57×10^{-7}	9.63×10^{-7}	1.78×10^{-6}
512	256	cpu	4.04×10^{-7}	4.04×10^{-7}	6.91×10^{-7}	1.18×10^{-6}	2.20×10^{-6}	4.19×10^{-6}
512	256	harley seal	7.59×10^{-7}	7.59×10^{-7}	1.05×10^{-6}	1.61×10^{-6}	1.72×10^{-6}	2.82×10^{-6}
512	256	lookup	2.98×10^{-7}	2.98×10^{-7}	4.64×10^{-7}	7.66×10^{-7}	1.40×10^{-6}	2.67×10^{-6}
512	256	lookup orig.	2.99×10^{-7}	2.99×10^{-7}	4.57×10^{-7}	8.01×10^{-7}	1.46×10^{-6}	2.81×10^{-6}
512	64	popcnt movdq	3.02×10^{-7}	3.02×10^{-7}	5.19×10^{-7}	9.99×10^{-7}	1.86×10^{-6}	3.60×10^{-6}
512	64	popcnt un. err.	4.36×10^{-7}	4.36×10^{-7}	7.09×10^{-7}	1.17×10^{-6}	2.06×10^{-6}	3.82×10^{-6}
256	256	cpu	1.95×10^{-7}	2.90×10^{-7}	5.32×10^{-7}	9.38×10^{-7}	1.81×10^{-6}	3.46×10^{-6}
256	256	harley seal	5.08×10^{-7}	6.06×10^{-7}	7.85×10^{-7}	1.16×10^{-6}	1.14×10^{-6}	1.85×10^{-6}
256	256	lookup	2.24×10^{-7}	3.13×10^{-7}	4.65×10^{-7}	5.71×10^{-7}	9.98×10^{-7}	1.83×10^{-6}
256	256	lookup orig.	2.15×10^{-7}	2.90×10^{-7}	4.56×10^{-7}	7.82×10^{-7}	1.44×10^{-6}	2.75×10^{-6}
256	64	popcnt movdq	1.60×10^{-7}	2.58×10^{-7}	4.73×10^{-7}	8.82×10^{-7}	1.73×10^{-6}	3.36×10^{-6}
256	64	popcnt un. err.	1.86×10^{-7}	3.08×10^{-7}	5.42×10^{-7}	1.04×10^{-6}	2.02×10^{-6}	3.93×10^{-6}

1. Two different vector widths for the bitwise *AND* operations: 256 and 512 bits.

2. Three different vector widths for the *popcount* operations: 64 bits, using the hardware *popcount* instruction from the Bit Manipulation Instructions (BMI) extension, and 256 and 512 bits, using the software implementations proposed by Muła et al.

3. Six different table row widths: 256, 512, 1024, 2048, 4096 and 8192 individuals in each row (or 32, 64, 128, 256, 512 and 1024 bytes per row, respectively), equal for cases and controls.

From these results we can conclude that the fastest implementation is dependant on the width of the genotype tables. For less than 512 individuals per subtable, the best times are obtained by the implementations that make use of the BMI *popcount* instruction. However, if we have more than 512 individuals, the lookup implementations for both the *AVX2* and *AVX512BW* extensions offer the fastest alternative for most of the widths tested. Taking a look at all the epistasis studies referenced throughout this thesis, we can find that most of them consider a number of individuals between 512 and 4096. Therefore, we will use the *AVX2* and *AVX512BW* implementations of the *popcount* lookup algorithm.

Vectorizing the `combine_and_popcount` function from Listing 4.3 requires vectorizing its auxiliary subroutine `popcnt_subtable`. Starting with the *AVX2* implementation, Listing 4.4 shows an implementation of the vectorized function, combining the computation of the new genotype table with the *popcount* lookup algorithm. This function includes the following modifications to the original lookup algorithm:

1. Instead of iterating over an input array as in the original popcount function (Lines 32–43 from file `popcnt-avx2-lookup.cpp`[2]), `popcnt_subtable` consists of three nested loops: the two outer ones (Lines 21 and 22) combining the different rows of the input genotype tables, and the two innermost loops (Lines 24 and 37) applying the *popcount* iteration to each 256-bit word of the two selected rows. The first of the two innermost loops (Lines 24–35) maintains the original unrolling of eigth 256-bit words.

2. Each iteration step (inlined function `iter`) reads a 256-bit word from each table row (Line 4), computes the bitwise *AND* of the two words (Line 5) and continues with the Muła et al. *popcount* algorithm (Lines 6–10, which correspond to Figure 10 from [90]).

Listing 4.5 shows the implementation of the same `popcnt_subtable` subroutine but using Intrinsics from the *AVX512BW* extension. The original *popcount* lookup algorithm for *AVX512BW* (file `popcnt-avx512bw-lookup.cpp`[3]) is very similar to its *AVX2* implementation, with the obvious difference of not applying unrolling to its innermost loop (Lines 39–49). Therefore, the same considerations for the *AVX2* implementation of the function apply to the *AVX512BW* algorithm: the function combines the input genotype tables using three nested loops (Lines 10, 11 and 14), and each *popcount* iteration of the Muła et al. algorithm is preceded by two loads that read a 512-bit word from each input genotype table (Lines 17 and 18) and a bitwise *AND* operation (Line 19).

[2] https://github.com/WojciechMula/sse-popcount/blob/master/popcnt-avx2-lookup.cpp
[3] https://github.com/WojciechMula/sse-popcount/blob/master/popcnt-avx512bw-lookup.cpp

Listing 4.4: Contingency table calculation function vectorized with *AVX2* Intrinsics

```
1  inline void iter(const uint64_t *ptr1, const uint64_t *ptr2, const __m256i &lu,
2    const __m256i &low_mask, __m256i &local)
3  {
4    __m256i o1 = _mm256_load_si256(ptr1), o2 = _mm256_load_si256(ptr2);
5    __m256i vec = _mm256_and_si256(o1, o2);
6    __m256i lo = _mm256_and_si256(vec, low_mask);
7    __m256i hi = _mm256_and_si256(_mm256_srli_epi16(vec, 4), low_mask);
8    __m256i popcnt1 = _mm256_shuffle_epi8(lu,lo);
9    __m256i popcnt2 =_mm256_shuffle_epi8(lu,hi);
10   local = _mm256_add_epi8(_mm256_add_epi8(local, popcnt1), popcnt2);
11 }
12 inline void popcnt_subtable(const uint64_t *gt_tbl1, const size_t size1,
13   const uint64_t *gt_tbl2, const size_t words, uint32_t *ct_tbl, size_t ct_size)
14 {
15   __m256i lookup = _mm256_setr_epi8(/*0*/0, /*1*/1, /*2*/1, /*3*/2, /*4*/1, /*5*/2,
16     /*6*/2, /*7*/3, /*8*/1, /*9*/2, /*a*/2, /*b*/3, /*c*/2, /*d*/3, /*e*/3, /*f*/4,
17     /*0*/0, /*1*/1, /*2*/1, /*3*/2, /*4*/1, /*5*/2, /*6*/2, /*7*/3, /*8*/1, /*9*/2,
18     /*a*/2, /*b*/3, /*c*/2, /*d*/3, /*e*/3, /*f*/4);
19   const __m256i low_mask = _mm256_set1_epi8(0xf);
20   size_t i, j, k;
21   for (i = 0; i < size1; ++i) {
22     for (j = 0; j < 3; ++j) {
23       __m256i acc = _mm256_setzero_si256();
24       for (k = 0; k + 32 <= words; k += 32) {
25         __m256i local = _mm256_setzero_si256();
26         iter(gt_tbl1+i*words+k+0, gt_tbl2+j*words+k+0, lookup, low_mask, local);
27         iter(gt_tbl1+i*words+k+4, gt_tbl2+j*words+k+4, lookup, low_mask, local);
28         iter(gt_tbl1+i*words+k+8, gt_tbl2+j*words+k+8, lookup, low_mask, local);
29         iter(gt_tbl1+i*words+k+12, gt_tbl2+j*words+k+12, lookup, low_mask, local);
30         iter(gt_tbl1+i*words+k+16, gt_tbl2+j*words+k+16, lookup, low_mask, local);
31         iter(gt_tbl1+i*words+k+20, gt_tbl2+j*words+k+20, lookup, low_mask, local);
32         iter(gt_tbl1+i*words+k+24, gt_tbl2+j*words+k+24, lookup, low_mask, local);
33         iter(gt_tbl1+i*words+k+28, gt_tbl2+j*words+k+28, lookup, low_mask, local);
34         acc = _mm256_add_epi64(acc, _mm256_sad_epu8(local, _mm256_setzero_si256()));
35       }
36       local = _mm256_setzero_si256();
37       for (; k < words; k += 4) {
38         iter(gt_tbl1+i*words+k, gt_tbl2+j*words+k, lookup, low_mask, local);
39       }
40       acc = _mm256_add_epi64(acc, _mm256_sad_epu8(local, _mm256_setzero_si256()));
41       ct_tbl[i*3+j] = _mm256_extract_epi64(acc,0) + _mm256_extract_epi64(acc,1) +
42         _mm256_extract_epi64(acc,2) + _mm256_extract_epi64(acc,3);
43   }}
44   for (i = size1 * 3; i < ct_size; ++i)
45     ct_tbl[i] = 0;
46 }
```

Listing 4.5: Contingency table calculation function vectorized with *AVX512BW* Intrinsics

```
1  inline void popcnt_subtable(const uint64_t *gt_tbl1, const size_t size1,
2    const uint64_t *gt_tbl2, const size_t words, uint32_t *ct_tbl,
3    const size_t ct_size)
4  {
5    const __m512i lookup = _mm512_setr_epi64(
6      0x0302020102010100llu, 0x0403030203020201llu, 0x0302020102010100llu,
7      0x0403030203020201llu, 0x0302020102010100llu, 0x0403030203020201llu,
8      0x0302020102010100llu, 0x0403030203020201llu);
9    const __m512i low_mask = _mm512_set1_epi8(0xf);
10   for (size_t i = 0; i < size1; ++i) {
11     for (size_t j = 0; j < 3; ++j) {
12       size_t k = 0;
13       __m512i acc = _mm512_setzero_si512();
14       while (k < words) {
15         __m512i local = _mm512_setzero_si512();
16         for (size_t l = 0; l < 255 / 8 && k < words; ++l, k += 8) {
17           __m512i z0 = _mm512_load_si512(gt_tbl2 + j * words + k);
18           __m512i z1 = _mm512_load_si512(gt_tbl1 + i * words + k);
19           __m512i z2 = _mm512_and_si512(z0, z1);
20           __m512i lo = _mm512_and_si512(z2, low_mask);
21           __m512i hi = _mm512_and_si512(_mm512_srli_epi32(z2, 4), low_mask);
22           __m512i popcnt1 = _mm512_shuffle_epi8(lookup, lo);
23           __m512i popcnt2 = _mm512_shuffle_epi8(lookup, hi);
24           local = _mm512_add_epi8(_mm512_add_epi8(local,popcnt1), popcnt2);
25         }
26         acc = _mm512_add_epi64(acc,_mm512_sad_epu8(local,_mm512_setzero_si512()));
27       }
28       ct_tbl[i * 3 + j] = _mm512_reduce_add_epi64(acc);
29   }}
30   for (size_t i = size1 * 3; i < ct_size; ++i)
31     ct_tbl[i] = 0;
32 }
```

4.1.3. Vectorization of the Mutual Information Calculation

Listing 4.6 shows a C++ function implementing the MI computation described in Section 1.3.3. It computes $H(X,Y)$ and $H(X)$ in a single *for* loop (Lines 6–13 and 14–17, respectively). The loop includes three *if* branches to avoid computing the logarithm of 0, which would lead to an undefined product of $0 \times -\infty$, resulting in a `NaN` value. $H(Y)$ and the inverse of the number of individuals (`iinds`) are provided as function arguments because they are independent of the genotype distribution of individuals, and thus can be calculated just once outside the function (Line 1). The MI function operates with `float` types since single-precision floating point numbers offer enough numerical precision to represent the MI values. The time complexity of this operation is $O(3^k)$, where k is the size of the combination represented in the input contingency table.

Listing 4.6: MI computation function

```
1  float MI(const ContingencyTable<uint32_t> &table,const float h_y,const float iinds)
2  {
3    float h_x = 0.0f, h_all = 0.0f;
4    const size_t table_size = table.size;
5    for (size_t i = 0; i < table_size; i++) {
6      float p_case = table.cases[i] * iinds;
7      if (p_case != 0.0f) {
8        h_all -= p_case * logf(p_case);
9      }
10     float p_ctrl = table.ctrls[i] * iinds;
11     if (p_ctrl != 0.0f) {
12       h_all -= p_ctrl * logf(p_ctrl);
13     }
14     float p_any = p_case + p_ctrl;
15     if (p_any != 0.0f) {
16       h_x -= p_any * logf(p_any);
17     }
18   }
19   return h_x + h_y - h_all;
20 }
```

In contrast to the two previous functions, calculating the MI of a contingency table requires floating-point arithmetic, including multiplications, Fused Multiply-Adds (FMAs) and logarithms. Multiplications and FMAs are supported natively, both for 256-bit and 512-bit vector operations, but there is no hardware instruction that implements a logarithm. However, Intel does provide an AVX logarithm routine through their Short Vector Math Library (SVML), an extension to the Intel Intrinsics available only with the Intel Compiler. GCC provides a vector implementation of the logarithm through GNU's *glibc* vector math library, available since version 2.22, although the number of vector functions available is much more limited compared to Intel's SVML.

Listing 4.7 shows a C++ function implementing the MI computation using AVX Intrinsics from the *AVX2* extension. This code assumes that the contingency table size is divisible by the vector unit width. Similar to the genotype table and contingency table calculations, we can achieve this by padding the input contingency table with 0's, which will not contribute to the final MI value. The computation follows the same strategy of avoiding the calculations of the logarithm of zero as in the regular MI implementation (Listing 4.6) but by different means: instead of skipping the logarithm altogether, which is not possible now unless all eight values of the vector are zero, the function replaces the zeros in the vector registers with values of one, which will evaluate to zero and will not contribute in the following FMA operations (Lines 10–11, 15–16

Listing 4.7: MI computation function vectorized with *AVX2* Intrinsics

```
1  float MI(const ContingencyTable<uint32_t> &table,const float h_y,const float iinds)
2  {
3    const __m256 ones = _mm256_set1_ps(1.0);
4    const __m256 ii = _mm256_set1_ps(iinds);
5    __m256 h_x = _mm256_setzero_ps();
6    __m256 h_all = _mm256_setzero_ps();
7    for (auto i = 0; i < table.size; i += 8) {
8      __m256i y0 = _mm256_load_si256(table.cases + i);
9      __m256 y3 = _mm256_mul_ps(_mm256_cvtepi32_ps(y0), ii);
10     __m256 y1 = _mm256_cmp_ps(y0, _mm256_setzero_ps(), _CMP_NEQ_OQ);
11     __m256 y4 = _mm256_log_ps(_mm256_blendv_ps(ones, y3, y1));
12     h_all = _mm256_fmadd_ps(y3, y4, h_all);
13     y0 = _mm256_load_si256(table.ctrls + i);
14     y4 = _mm256_mul_ps(_mm256_cvtepi32_ps(y0), ii);
15     __m256 y2 = _mm256_cmp_ps(y0, _mm256_setzero_ps(), _CMP_NEQ_OQ);
16     __m256 y5 = _mm256_log_ps(_mm256_blendv_ps(ones, y4, y2));
17     h_all = _mm256_fmadd_ps(y4, y5, h_all);
18     y5 = _mm256_add_ps(y3, y4);
19     y1 = _mm256_or_ps(y1, y2);
20     y3 = _mm256_log_ps(_mm256_blendv_ps(ones, y5, y1));
21     h_x = _mm256_fmadd_ps(y5, y3, h_x);
22   }
23   y3 = _mm256_hadd_ps(h_all, h_x);
24   return (y3[0] + y3[1] + y3[4] + y3[5]) - h_y - (y3[2] + y3[3] + y3[6] + y3[7]);
25 }
```

and 19–20).

Moving onto *AVX512BW*, this extension provides mask registers and masked operations, which allow for the execution of operations only on some of the values contained in the vector register. Masked logarithms are a very convenient operation to skip the computation of the logarithm of zero. With masks we can avoid the zeros without having to blend two vector registers beforehand. Unfortunately, masked logarithms are only available under the SVML and for a vector width of 512 bits. The rest of the implementations still have to rely on the sequence of blends and logarithms.

Listing 4.8 shows the same code implemented using 512-bit intrinsics from the *AVX512BW* extension. Comparisons are now made using the new intrinsic functions operating with 16-bit masks (Lines 10, 15 and 19) instead of a whole vector register, and the blend operation takes these masks as an argument. If the SVML is available, the logarithm plus blend sequences of operations (Lines 11, 16 and 20) can be replaced with a single call to the intrinsic `mm512_mask_log_ps`, which only calculates the logarithm on the positions specified by the mask.

The *AVX512BW* extension also includes mask functions for 256-bit operations.

Listing 4.8: MI computation function vectorized with *AVX512BW* Intrinsics

```
1  float MI(const ContingencyTable<uint32_t> &table,const float h_y,const float iinds)
2  {
3    const __m512 ones = _mm512_set1_ps(1.0);
4    const __m512 ii = _mm512_set1_ps(inv_inds);
5    __m512 h_x = _mm512_setzero_ps();
6    __m512 h_all = _mm512_setzero_ps();
7    for (auto i = 0; i < table.size; i += 8) {
8      __m512i z0 = _mm512_load_si512(table.cases + i);
9      __m512 z2 = _mm512_mul_ps(_mm512_cvtepi32_ps(z0), ii);
10     __mmask16 m1 = _mm512_cmp_epi32_mask(z0, _mm512_setzero_si512(), _MM_CMPINT_NE);
11     __m512 z3 = _mm512_log_ps(_mm512_mask_blend_ps(m1, ones, z2));
12     h_all = _mm512_fmadd_ps(z2, z3, h_all);
13     z0 = _mm512_load_si512(table.ctrls + i);
14     z3 = _mm512_mul_ps(_mm512_cvtepi32_ps(z0), ii);
15     __mmask16 m2 = _mm512_cmp_epi32_mask(z0, _mm512_setzero_si512(), _MM_CMPINT_NE);
16     __m512 z4 = _mm512_log_ps(_mm512_mask_blend_ps(m2, ones, z3));
17     h_all = _mm512_fmadd_ps(z3, z4, h_all);
18     __m512 z1 = _mm512_add_ps(z2, z3);
19     __mmask16 m3 = _kor_mask16(m1, m2);
20     z2 = _mm512_log_ps(_mm512_mask_blend_ps(m3, ones, z1));
21     h_x = _mm512_fmadd_ps(z1, z2, h_x);
22   }
23   return _mm512_reduce_add_ps(h_all) - _mm512_reduce_add_ps(h_x) - h_y;
24 }
```

Therefore, for comparison purposes, a third version of the MI function using a width of 256 bits was also created. This version uses the same sequence of blend plus logarithm intrinsic functions shown in Listing 4.8 both for the SVML and *glibc* libraries, since Intel does not include in the SVML a masked version of the 256-bit logarithm intrinsic.

4.2. Vector-Aware Exhaustive Epistasis Search Algorithm

This section presents an exhaustive epistasis search algorithm supporting interactions of any given order (Section 4.2.1) that makes use of the vector implementation of the MI association test presented in the previous section. From this algorithm, further improvements are introduced in order to mitigate the runtime penalties caused by the differences in frequency at which different vector operations run in *x86_64* processors (Section 4.2.2).

Algorithm 4.1: `non-segmented_sequential_search`: Any-order exhaustive exploration of all variant combinations

Input:
a: Array of genotype tables representing n input SNPs for m individuals
k: Order of the interactions to identify
Output:
List of k-SNP combinations and their associated MI value

```
1  gt ← Array of k − 1 genotype tables, for sizes between 1 and k − 1
2  ct ← Contingency table of size k
3  s ← Empty stack
4  e ← H(Y)
5  inv_inds ← 1/m
6  out ← Empty list
7  for i ← 0 to n do
8  |   gt[1] ← a[i]
9  |   for j ← i + 1 to n do
10 |   |   push(s, {i, j})
11 |   while s is not empty do
12 |   |   {c_1, ..., c_l} ← pop(s)
13 |   |   if l < k then
14 |   |   |   gt[l] ← combine(gt[l − 1], a[c_l])
15 |   |   |   for j ← c_l + 1 to n do
16 |   |   |   |   push(s, {c_1, ..., c_l, j})
   |   |   else
17 |   |   |   ct ← combine_and_popcnt(gt[l − 1], a[c_l])
18 |   |   |   v ← MI(ct, e, inv_inds)
19 |   |   |   append(out, {{c_1, ..., c_l}, v})
20 return out
```

4.2.1. Sequential Exhaustive Algorithm

Algorithm 4.1 shows the pseudocode of a depth-first exploration algorithm which relies on the genotype table, contingency table and MI functions previously defined to combine the different SNPs of the input data and assess the degree of association between the SNP combinations and the phenotype of study. The key element of this algorithm is that it iterates over all the combinations in a depth-first manner with the help of a stack. This is fundamental to prevent multiple calculations of the same genotype table, since combinations may share a common set of SNPs with other combinations. When the combination space is explored depth-first, we exhaust all combinations starting with a particular prefix (and therefore its corresponding genotype table) before moving onto the next one.

The arguments to this routine are an array a containing the genotype tables of all individual SNPs in the data and the order k of the interactions to locate. As a result, it returns and a list of k-SNP combinations and the MI associated with each one. In the first four lines, the function starts by allocating enough space for an array gt of $k-1$ genotype tables of size 1 to $k-1$, a contingency table ct for combinations of the target size k, a stack s of combinations of SNP indexes and a list *out* in which the output combinations and MI values are stored. Before starting the exploration, the function computes the inverse of the number of individuals *inv_inds*, and the entropy e of the phenotype variability $H(Y)$ (Lines 4 and 5), the two arguments of the MI function common to all combinations.

After that, the function starts to loop through all SNPs, exploring all the combinations starting with that SNP before moving onto the next one. To do this, the genotype table of the SNP i is copied in $gt[1]$, and all combinations of two SNPs starting with that one are pushed into the stack (Lines 8–10). Then, using a *while* loop, the combinations of the stack are processed until it is emptied. In each iteration, the top combination $\{c_1,\ldots,c_l\}$ of the stack is popped (Line 12). If l is smaller than the target interaction order k, its corresponding genotype table is computed from the genotype table of its prefix (stored in the array gt) and the table of the last SNP $a[c_l]$ (Line 14). Then, all subsequent combinations starting with $\{c_1,\ldots,c_l\}$ are pushed into the stack (Lines 15–16). Otherwise, if l is equal to k, its contingency table and MI are computed, and the result is stored in the vector of results *out* (Lines 17–19).

For simplicity, in the pseudocode, all combinations with its MI are appended to the vector of results, although in the actual implementation only the combinations with the highest MI value are retained in the vector. The computational time complexity of this algorithm is $O((3n)^k m)$, where n and m are the number of SNPs and samples per SNP in the data, respectively, and k is the size of the combinations explored.

4.2.2. Vector-Aware Sequential Exhaustive Algorithm

Although the algorithm presented in Algorithm 4.1 already incorporates the SIMD functions described in the previous subsections, it does not consider all the implications that vectorization brings with it. The performance per core is being penalized due to the interleaved execution of vector instructions running at very different fre-

quencies. Intel CPUs, such as the Intel 6240 used during the evaluation, are known to downscale their CPU clock frequency based on the number of active cores and the sequence of instructions executed due to differences in power consumption and/or heat dissipation. In the processor technical document [91], Intel identifies three different frequency licenses in which the processor operates: non-AVX, AVX 2.0 and AVX-512 base core frequencies. Furthermore, different operations inside each license are not guaranteed to run at the same frequency, these are only base frequencies that the processor is guaranteed to run at. For example, floating-point arithmetic vector operations run at a slower clock frequency than integer arithmetic or bitwise vector operations.

As a direct consequence of this, the exploration algorithm would run on the lowest frequency imposed by any of the vector operations, since the change in frequency is not immediate and depends on the pipeline of operations executed. To resolve it, Algorithm 4.2 proposes a modification to the previous one, segmenting the operations into different blocks attending at the running frequencies to avoid the frequency change problem.

Instead of declaring a single contingency table, the function now reserves space to store b combinations and compute b tables before applying MI to any of them (Lines 2–3). Combinations are now explored using two nested *while* loops, the outer one iterating until the stack is empty and all combinations starting with the SNP i have been explored (Line 9), and the inner one iterating until the block of b contingency tables has been filled (Line 11).

Every iteration of the innermost loop starts by checking if the stack is empty. If that is the case, and there are no more SNPs to explore, the loop exits (Line 14); otherwise, the genotype table of the SNP i is copied in $gt[1]$, all combinations of two SNPs starting with i are pushed into the stack, the counter i is increased and the execution continues on the next iteration of the inner while loop (Lines 15–19). If the stack is not empty, the loop operates similarly as the old one: the top combination $\{c_1,\ldots,c_l\}$ of the stack is popped (Line 20). If l is smaller than the target interaction order k, its genotype table is computed and stored in the array of genotype tables gt, and all subsequent combinations starting with $\{c_1,\ldots,c_l\}$ are pushed into the stack (Lines 22–24). Otherwise, $\{c_1,\ldots,c_l\}$ and its contingency table are stored in the *ids* and *ct* arrays, respectively (Lines 25–27). When the arrays *ids* and *ct* of b index combinations and ta-

Algorithm 4.2: `segmented_sequential_search`: Any-order exhaustive exploration of all variant combinations, with vector operations divided into two blocks

Input:
a: Array of genotype tables representing n input SNPs for m inds.
k: Order of the interactions to identify
b: Size of the block of operations

Output:
List of k-SNP combinations and their associated MI value

```
1  gt ← Array of k − 1 genotype tables, for sizes between 1 and k − 1
2  ids ← Array of b combinations of size k
3  ct ← Array of b contingency tables of size k
4  s ← Empty stack
5  e ← H(Y)
6  inv_inds ← 1/m
7  out ← Empty list
8  i ← 0
9  while s is not empty or i < n do
10     cnt ← 0
11     while cnt < b do
12         if s is empty then
13             if i ≥ n then
14                 Break from the inner while loop
               else
15                 gt[1] ← a[i]
16                 for j ← i + 1 to n do
17                     push(s, {i, j})
18                 i ← i + 1
19                 Continue on the next iteration of the inner while loop
20         {c_1, ..., c_l} ← pop(s)
21         if l < k then
22             gt[l] ← combine(gt[l − 1], a[c_l])
23             for j ← c_l + 1 to n do
24                 push(s, {c_1, ..., c_l, j})
           else
25             ids[cnt] ← {c_1, ..., c_l}
26             ct[cnt] ← combine_and_popcnt(gt[l − 1], a[c_l])
27             cnt ← cnt + 1
28     for j ← 0 to cnt do
29         v ← MI(ct[j], e, inv_inds)
30         append(out, {ids[j], v})
31     return out
```

bles, respectively, have been filled, the inner *while* finishes and a *for* loop iterates over all the computed contingency tables, calculating its MI and adding the combination into the vector of results v (Lines 28–30).

The selection of a proper value for the block size b is key in order to obtain good performance. It has to be large enough to make the impact of the transition between frequencies negligible, but not large enough to exceed the second-level cache of the processor. Through experimental testing, we found that an appropriate b for the Intel 6240, the processor used in the evaluation, is $1\,474\,560/3^k$, with k being the order of the search. This size corresponds to the smallest block size tested at which the average running frequencies of the functions are very close or equal to the running frequencies of these same functions in isolation.

4.3. Evaluation

We have conducted an extensive evaluation of the performance achieved by the automatic vectorization offered by the GCC and Intel compilers, in contrast to manual vectorization using Intel Intrinsics, when implementing a SIMD epistasis detection algorithm. It considers the performance of the different functions that compose the epistasis search in isolation, as well as the whole depth-first search algorithm. This evaluation starts by assaying the individual functions separately, and identifying which of the implementations obtains the best performance in each part. Then, the search algorithm is evaluated showcasing how the relative differences in time spent in each of the functions, and the operations that each function involves, influence the performance of the whole search. At last, the best performing implementation is compared against the original *MPI3SNP* [2] program using the compiler's automatic vectorization, to put into perspective the performance gain achieved.

Given the exponential time complexity of the operations that compose an epistasis search, and the search itself, it is difficult to represent elapsed time results for different problem sizes in the same graph and extract conclusions from them. For this reason, this evaluation uses the average elapsed time per cell or row (depending on the computation being evaluated) as the metrics to present the results. These measures of time express the compute time relative to the complexity of the computation, thus remov-

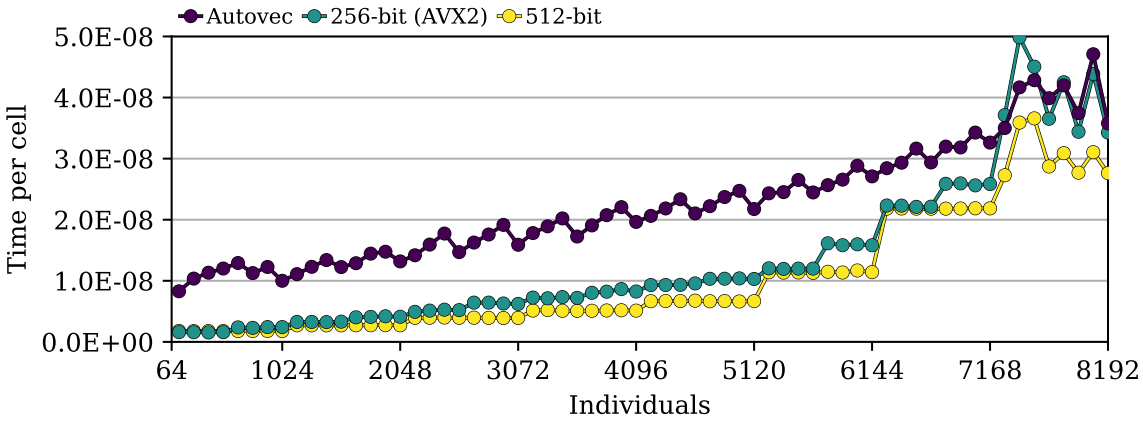
Table 4.2: Characteristics of the *SCAYLE* nodes from the `cascadelake` partition.

	SCAYLE NODE (CASCADELAKE PARTITION)
CPUs	2x Intel 6240 (36 cores) @ 2.6 GHz
MEMORY	192 GB
NETWORK	Infiniband HDR @ 100 Gbps
GPU CARDS	NVIDIA V100
OS	CentOS 7.7
KERNEL	3.10

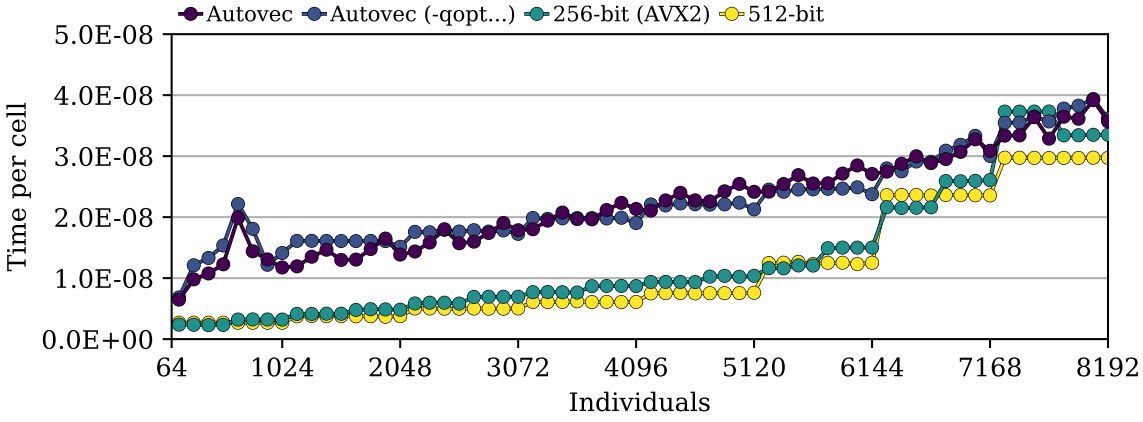
ing the impact of this complexity from the results and highlighting the differences in performance from multiple implementations of the same operation.

The two compilers used throughout the evaluation are GCC version 8.3 (with *glibc* version 2.29) and ICC version 19.1. The same optimization flags were used for both compilers: `-O3`, `-march=native` and `-mtune=native`. Additionally, we enabled optimizations on floating-point arithmetic operations using `-fast-math` and `-fp-model=fast` for the two compilers respectively, as it is a requirement for GCC in order to vectorize some calls to the math library. Furthermore, for the automatic vectorization, we considered the effects of indicating a preference for a particular vector width during the compilation through the flags `-mprefer-vector-width={256,512}` for the GCC compiler and `-qopt-zmm-usage={low,high}` for the Intel compiler. Only the flag `-qopt-zmm-usage=high` had a positive impact on performance, thus it is the only one included in the results.

All experiments were run on the `cascadelake` partition of the *SCAYLE* cluster, briefly described in Table 4.2. As mentioned in Section 4.2.2, performance during SIMD operation in modern Intel CPUs is tied to the number of active threads and the type of vector operations used in the instruction pipeline [91]. Therefore, to obtain a realistic multithreaded performance, elapsed times throughout the evaluation are measured during a simultaneous execution of the function in question on every core of the processor. The 18 different times are then averaged and presented as a single value.



(a) GCC compiler



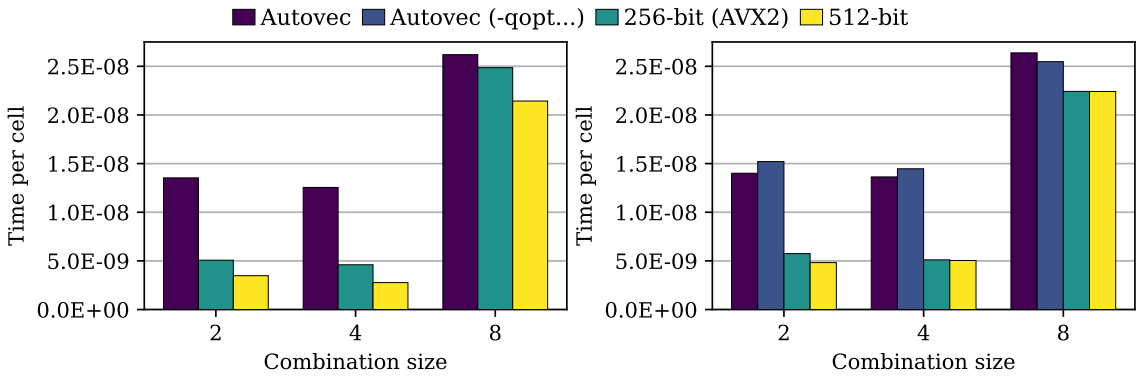
(b) ICC compiler

Figure 4.1: Average elapsed time per row during the calculation of genotype tables, for an increasing number of individuals and a fixed combination size of three, both using the GNU C Compiler and the Intel C++ Compiler.

4.3.1. Genotype Table Calculation Performance

Figs. 4.1 and 4.2 represent the performance results for the genotype table calculation function. The figures compare the performance of the explicit vectorization using 256-bit and 512-bit vector instructions with the automatically vectorized code, both for the GCC and Intel compilers. The measure of time used in both figures is the average time per row, that is, the average elapsed time during the calculation of a single row of the table including both cases and controls, for all of the genotype tables of the order and number of individuals specified.

Both compilers are capable of automatically vectorizing this function with no prob-



(a) GCC compiler (b) ICC compiler

Figure 4.2: Average elapsed time per row during the calculation of genotype tables, for combination sizes of 2, 4 and 8 and a fixed number of 2048 individuals, both using the GNU C Compiler and the Intel C++ Compiler.

lems. Despite this, and as the figures show, the performance of the autovectorization for both compilers is worse than the performance of both explicitly vectorized alternatives.

Fig. 4.1 represents the time per row during the computation of genotype tables corresponding to a combination of three SNPs, for a growing number of individuals. The time per row grows linearly with the number of individuals, as every row of the genotype table contains information about all the individuals in the data. Both compilers show a gap between the performance of the automatic and explicit vectorizations that is present until a number of individuals higher than 7040. The 512-bit explicit implementation performs slightly better in general than the 256-bit one.

Fig. 4.2 represents the time per row during the computation of genotype tables corresponding to combinations of 2, 4 and 8 SNPs, using a fixed number of individuals of 2048. Here, the time per row should remain constant when increasing the size of the combinations (k), as the number of rows in the table (3^k) grows with the number of SNPs in combination considered, but each row contains the same 2048 individuals. This is the case for combination sizes smaller than eight, where the time per row remains mostly constant between combination sizes of two to seven. Starting at genotype tables of eight SNPs, there is an increase in the elapsed time due to the size of the operands and result tables exceeding the level 1 data cache of the processor, which is manifested in the results by bringing the vector and non-vector performances much

closer.

For second and fourth-order interactions, both explicit vectorization alternatives obtain again better results than the vectorization applied by the compiler, with the 512-bit implementation performing the best. For eighth order interactions the performance gap is smaller, with less relative difference between implementations.

When taking a look at the frequencies at which the different implementations run, we observe that the genotype table calculation runs at 3270 MHz for the 256-bit vector width and 2805 MHz for the 512-bit vector width. In a different architecture, or in future Intel microarchitectures, where the difference in frequencies between vector widths could be smaller or nonexistent, we can expect the performance gap between the two widths to be larger.

As an example, the elapsed time per row of calculating a fourth-order genotype table of 2048 individuals at a fixed frequency of 2.6 GHz (the base frequency of the processor) is 6.52×10^{-9} s and 3.00×10^{-9} s for the explicit 256-bit and 512-bit implementations under GCC, respectively; and 6.32×10^{-9} s and 4.04×10^{-9} s for the same implementations under Intel, respectively. That is, the 512-bit implementation is 2.18 and 1.56 times faster than the 256-bit one for each compiler, significantly larger than what we observe in Fig. 4.2 between these two implementations (1.66 and 1.01).

4.3.2. Contingency Table Calculation Performance

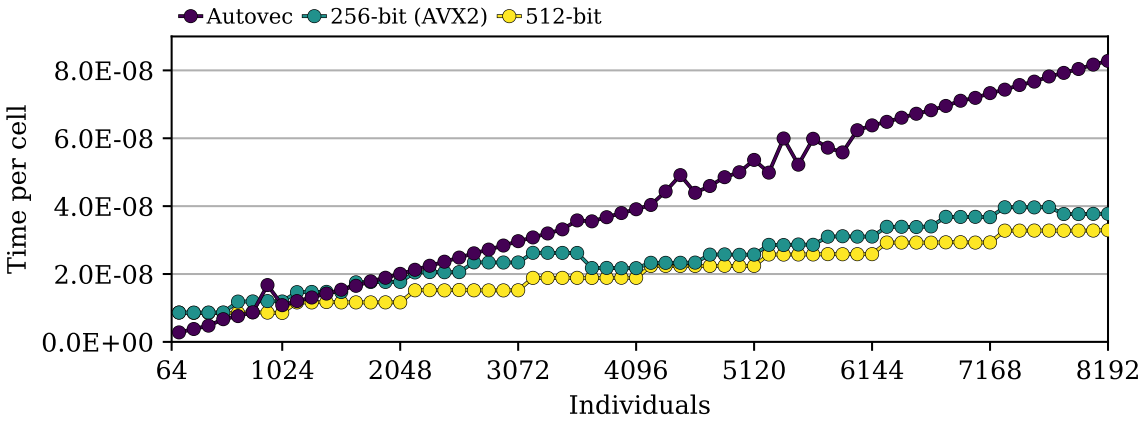
Figs. 4.3 and 4.4 represent the performance results for the contingency table calculation function. Similar to the figures from the genotype table calculation, these also compare the performance of the explicit vectorization using 256-bit and 512-bit vector operations with the automatically vectorized code using both GCC and Intel compilers. In this case, we use the average elapsed time per cell to represent the performance results, that is, the average time for the calculation of a single cell of the table, for all the contingency tables of the order and number of individuals specified.

For this function, only the Intel compiler is capable of vectorizing the *popcount* operation via the introduction of its own vector implementation. GCC, on the other hand, refuses to vectorize this function due to the presence of the aforementioned operation inside the innermost loop.

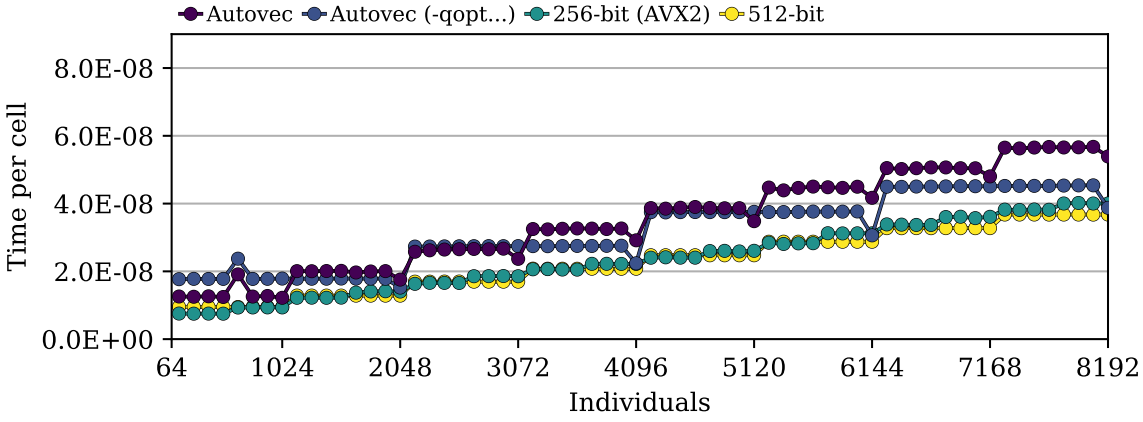
Fig. 4.3 represents the time per cell during the computation of contingency tables corresponding to a combination of three SNPs, for a growing number of individuals. The elapsed time per cell during the creation of contingency tables also grows linearly with the number of individuals, since the function operates with the rows from the two previous genotype tables, which include the data of all individuals. The differences in compiler behavior are apparent: GCC results display a linear increase of the elapsed time per cell with the number of individuals, at a faster pace than the Intel results do due to the lack of vectorization. It is also worth noting that there is a small reduction of the elapsed time per cell in the explicit 256-bit vectorization under GCC at 3712 individuals, which corresponds to the minimum number of individuals required to enter the loop that includes unrolling (Listing 4.4, Line 24). Anyhow, both explicit implementations are faster than the codes that the two compilers offer, with the 512-bit explicit vectorization being the fastest alternative.

Fig. 4.4 represents the time per cell during the computation of contingency tables for combination sizes of 2, 4 and 8, and using the same number of 2048 individuals. Analogous to the elapsed time per row during the calculation of genotype tables, the time per cell should also remain constant with the size of the combinations explored. However, contrary to those results, there is no increase in the elapsed time for eight order tables due to cache problems thanks to the avoidance of the genotype table storage. This is due to the merge of the last level genotype table calculations and contingency table computations in a single function. Results show that the explicit implementations are faster than the compiler-generated code, with the 512-bit vectorization being the fastest implementation.

Similar to the genotype table calculations, we observe that the contingency table computation function runs at a frequency of 3195 MHz for the 256-bit vector width and 2800 MHz for the 512-bit vector width. If we run the function at a fixed frequency of 2.6 GHz (the base frequency of the processor), the elapsed time per cell of calculating a fourth-order contingency table of 2048 individuals is 1.97×10^{-8} s and 1.24×10^{-8} s for the explicit 256-bit and 512-bit implementations under GCC, respectively; and 1.58×10^{-8} s and 1.29×10^{-8} s for the same implementations under Intel, respectively. This represents a speedup of 1.59 and 1.23 between vector widths for each compiler, slightly larger than those observed in Fig. 4.4 (1.42 and 1.11).



(a) GCC compiler

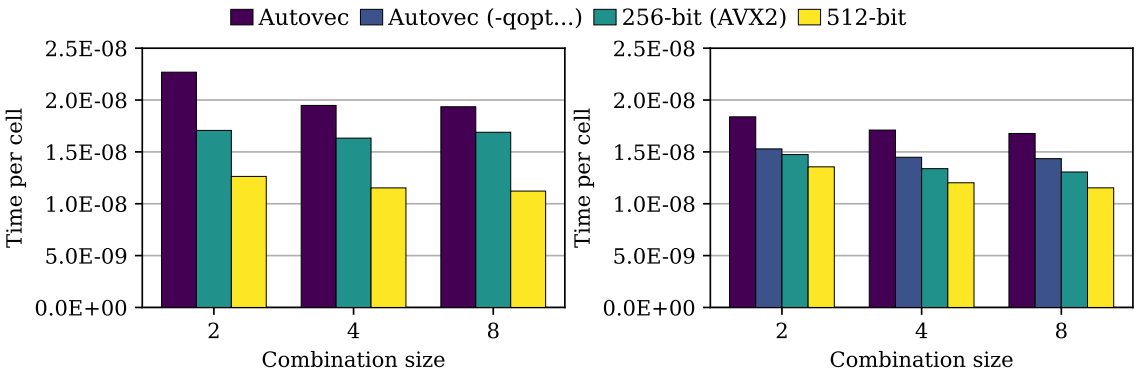


(b) ICC compiler

Figure 4.3: Average elapsed time per cell during the calculation of contingency tables, for an increasing number of individuals and a fixed combination size of three, both using the GNU C Compiler and the Intel C++ Compiler.

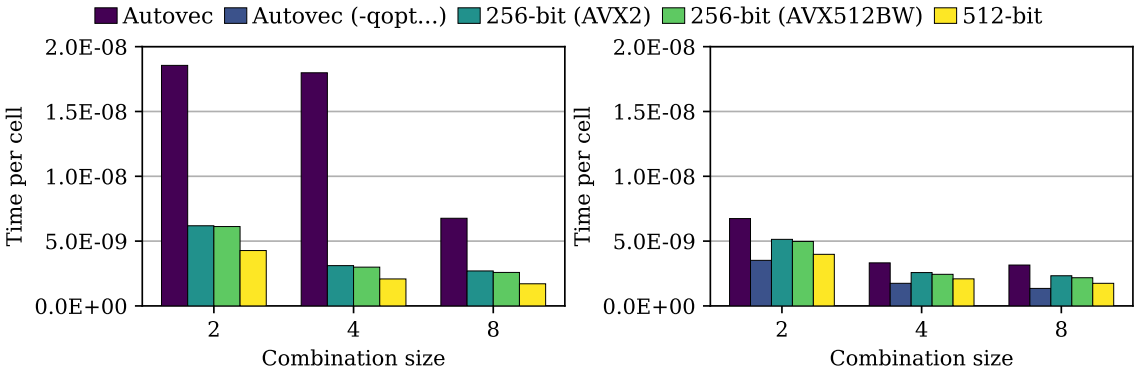
4.3.3. Mutual Information Calculation Performance

Fig. 4.5 shows the performance results for the MI computation function. This figure compares the performance of the automatically vectorized code with three explicit implementations: two 256-bit vector implementations using Intrinsics from the *AVX2* and *AVX512BW* extensions, respectively, and a 512-bit vector implementation using Intrinsics from the *AVX512BW* extension. Here, we also use the time per cell to represent the performance results. This time measures the average elapsed time during the MI calculations corresponding to a single cell of the contingency table (both for cases and controls), for all the tables of the specified order. The number of cells of a



(a) GCC compiler (b) ICC compiler

Figure 4.4: Average elapsed time per cell during the calculation of contingency tables, for combination sizes of 2, 4 and 8 and a fixed number of 2048 individuals, both using the GNU C Compiler and the Intel C++ Compiler.



(a) GCC compiler (b) ICC compiler

Figure 4.5: Average elapsed time per cell during the calculation of the MI metric, for combination sizes of 2, 4 and 8, both using the GNU C Compiler and the Intel C++ Compiler.

contingency table only depends on the number of SNPs in combination. Thus, MI, as opposed to the previous routines, does not depend on the number of individuals.

Results show that the time per cell for the explicit vector implementations generally decreases with the table size, despite the fact that, ideally, the workload per contingency table cell should remain constant regardless of the size of the table. This can mostly be attributed to the additional computations derived from the padding introduced in the input contingency tables. The larger the tables are, the lower the number of unnecessary computed cells is relative to the total number of cells in the table.

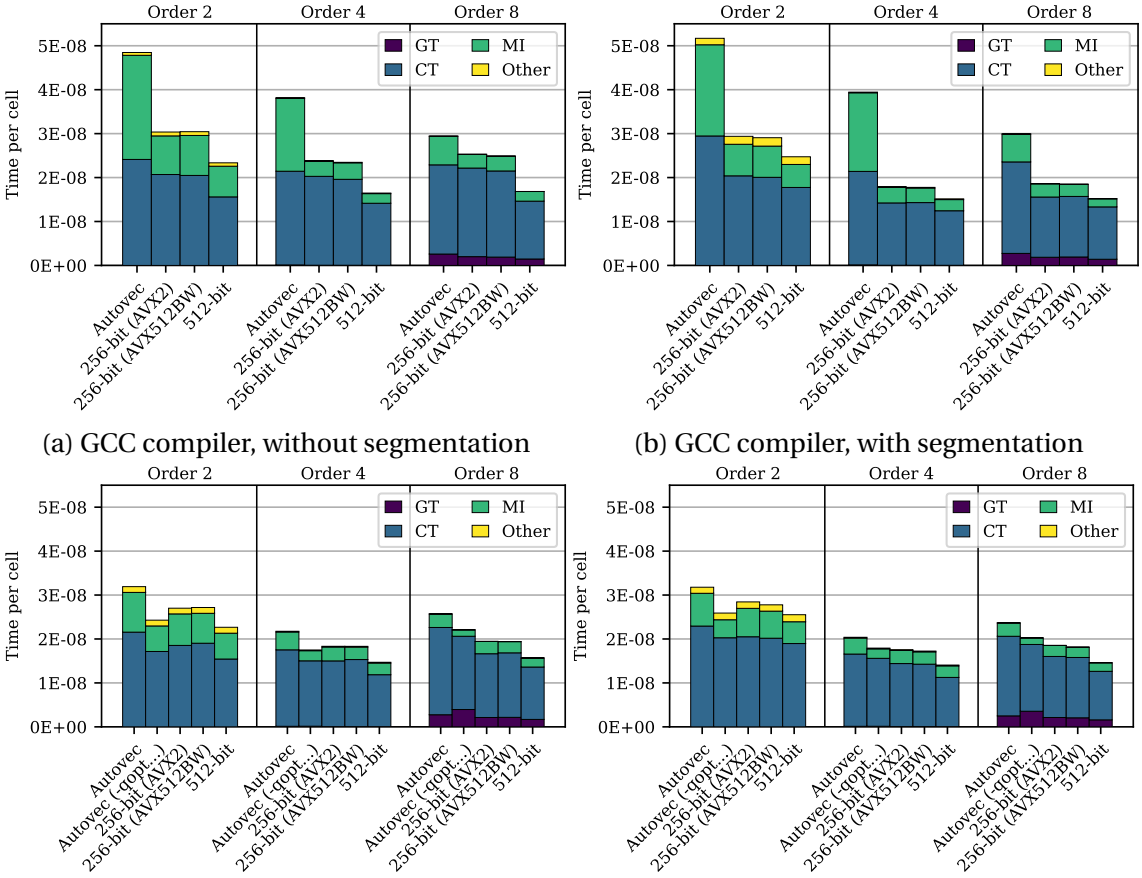
The best vector performance is achieved by the automatic vectorization of the Intel Compiler when coupled with the flag `-qopt-zmm-usage=high`. In contrast, GCC's automatic vectorization does not vectorize the loop, despite having a vectorized logarithm function available in the *glibc* library. Explicit vectorization using 512-bit AVX instructions obtains the best performance out of the explicit vector implementations (for GCC it is the fastest alternative). Furthermore, the introduction of 256-bit *AVX512BW* masked instructions in the function have no significant impact on the elapsed time when compared to the *AVX2* implementation.

When examining the assembly code to characterize the difference in performance between the explicit vector implementations and the code that the Intel auto-vectorizer generates, we found that the compiler calls a function from the SVML that is not available using intrinsics: `__svml_logf8_mask_e9` (a logarithm function for a vector width of 256 bits that uses a masked input). Therefore, in some scenarios, explicit vectorization may never obtain a performance equal or better than Intel's auto-vectorization due to the difference in SVML functions available through intrinsics.

As for the frequencies at which the function is executed, the 256-bit implementation runs at 2805 MHz (we only measured the 256-bit implementation using *AVX2* Intrinsics, since the elapsed times are almost the same) while the 512-bit implementation runs at 2500 MHz. These frequencies are considerably lower than the two previous functions due to the usage of floating-point arithmetic operations. If the running frequencies are fixed to 2.5 GHz (slightly lower than the base frequency because the 512-bit implementation runs at this frequency), we observe that the elapsed time per cell of calculating the MI of a fourth-order contingency table are 3.48×10^{-9} s and 2.10×10^{-9} s for the 256-bit and 512-bit vectorizations under GCC, respectively, and 2.87×10^{-9} s and 2.10×10^{-9} s under the Intel Compiler. This represents a speedup of 1.66 and 1.36 between vector widths for each compiler, respectively, slightly larger than those observed in Fig. 4.5 (1.49 and 1.23).

4.3.4. Exhaustive Search Performance

At last, Figs. 4.6 and 4.7 present the performance results for the whole exhaustive search algorithm. The two figures compare the performance of the 256 and 512-bit explicit vectorization approaches using operations from the *AVX2* and *AVX512BW* ex-



(a) GCC compiler, without segmentation

(b) GCC compiler, with segmentation

(c) ICC compiler, without segmentation

(d) ICC compiler, with segmentation

Figure 4.6: Average elapsed time per cell during the exhaustive search of epistasis, for combination sizes of 2, 4 and 8 and a fixed number of 2048 individuals, both using the GCC and Intel compilers. The times for each approach is divided into the calculation of the Genotype Tables (GT), Contingency Tables (CT), Mutual Information (MI) and the rest of the operations included in the algorithm.

tensions, with the automatically vectorized code using both GCC and Intel compilers, for both versions of the search algorithm presented in Algorithms 4.1 and 4.2.

Figs. 4.6 and 4.7 use the average elapsed time per cell to represent the performance results. The time per cell is the average elapsed time spent during the computation of a single contingency table cell, the subsequent MI operations corresponding to the cell of the table and a fraction of the time spent during the calculations of previous genotype tables (this time is equally divided across all cells of all combinations that make use of that genotype table), for all of the contingency tables of the order, number

Table 4.3: Problem sizes used during the exhaustive search evaluation. The number of combinations is the binomial of the size of the combinations and the number of SNPs. The number of cells is the product between the number of combinations and the number of cells (3^k). The deviation is the relative difference of the number of cells from the number used for the second-order.

SIZE	SNPS	COMBINATIONS	TOTAL CELLS	DEVIATION
2	25000	312487500	2812387500	0.00 %
4	242	34389810	2785574610	−0.95 %
8	23	490314	3216950154	14.39 %

of SNPs and individuals specified.

Fig. 4.6 represents the time per cell, shown as stacked bars indicating the fraction of the time spent in each of the functions, during the search of epistasis in combinations of 2, 4 and 8 SNPs, and for a fixed number of individuals of 2048. The number of SNPs was tied to the size of the combinations so that the workload among different explorations was as similar as possible. Table 4.3 indicates, for each exploration order, the number of SNPs selected, the resulting number of SNP combinations of said order, the number of different cells among those combinations and the difference in workload that that exploration order and number of SNPs supposes from the first one. The figure shows that the 512-bit explicit vectorization performs the best out of all of the versions compared, which is coherent with what we saw during the evaluation of the individual functions. The 256-bit explicit implementations obtain practically the same results and the only compiler-generated vectorization that beats any of the explicit vectorizations is the Intel Compiler when coupled with the optimization flag `-qopt-zmm-usage=high` for low-order epistasis searches.

The segmentation of operations introduced in the algorithm has an overall positive effect on the explicitly vectorized implementations. From a CPU frequency perspective, the segmentation algorithm achieves its goal. When there is no separation between integer/binary arithmetic and floating-point arithmetic, the whole program runs at 2.8 and 2.5 GHz for the 256-bit and 512-bit implementations, respectively. However, when there is segmentation, genotype and contingency table calculations run at 3.05 and 2.75 GHz, and the MI operations run at 2.8 and 2.5 GHz for each implementation respectively. From the performance perspective, the segmentation strategy only works with the explicitly vectorized implementations and results in a noticeable reduc-

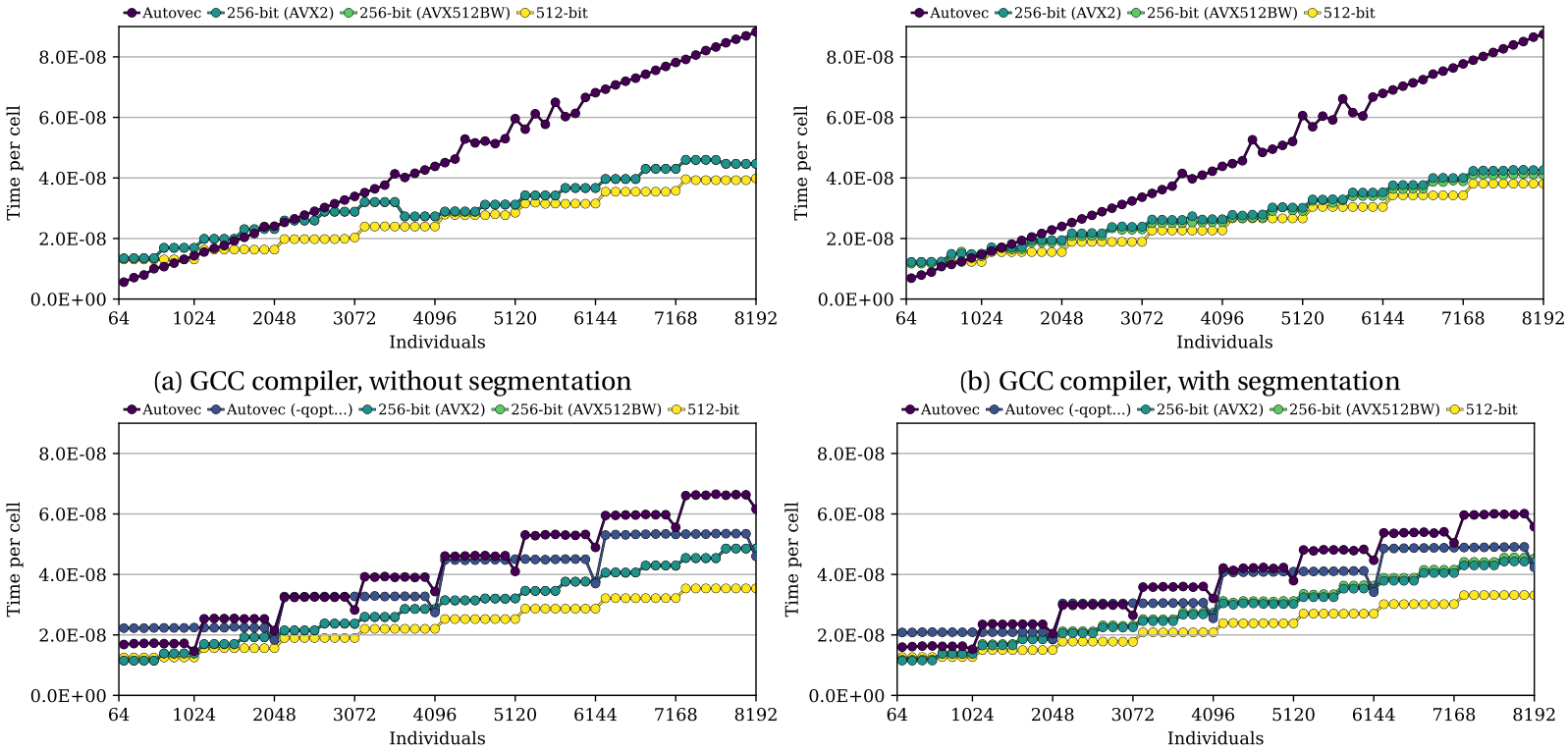
tion of the elapsed time of the searches under GCC, and a much smaller gain under the Intel Compiler for high-order interactions.

Fig. 4.7 represents the time per cell for a growing number of individuals from 128 to 8192, using a fixed combination size of three and a fixed number of SNPs of 680. Although the number of individuals is irrelevant to the calculation of the MI, it affects the calculation of the genotype tables and contingency tables, and therefore the time per cell during the whole search also grows linearly with the number of individuals, although with a less pronounced slope than the two first individual operations. These two subfigures show similar behaviour to the one shown in Fig. 4.3 because the calculation of the contingency table accounts for the majority of the elapsed time during the whole search.

Results from Fig. 4.7 show that the explicit vectorization using 512-bit operations achieve the best times. These are very similar to those of the contingency table calculation function (Fig. 4.3), which makes sense since it is the most time-consuming function of the algorithm as Fig. 4.6 showed. The best implementation is again the explicit 512-bit vectorization.

4.3.5. Performance of the Vectorized Search Compared Against MPI3SNP

To conclude the evaluation, Table 4.4 compares the elapsed time required to complete a third-order epistasis search using the original *MPI3SNP* [2] program and the explicit 512-bit vectorization proposed in this chapter, for an input data consisting of 1000 and 4000 SNPs and 1000 and 2000 individuals, and using a single core of the processor. *MPI3SNP* was compiled using the same flags indicated during the introduction of this section, enabling the automatic vectorization in both compilers. Results show that the vector implementation of the algorithm speeds the execution up by an average factor of 7 using GCC and 12 using the Intel Compiler. The speedup with Intel is in part due to its poor memory handling when allocating memory for objects inside a loop, something that has been accounted for in the new implementation and that *MPI3SNP* does not do. Therefore, we believe that the speedups obtained by GCC paint a more realistic picture of what speedups should be expected of this algorithm.



(a) GCC compiler, without segmentation

(b) GCC compiler, with segmentation

(c) ICC compiler, without segmentation

(d) ICC compiler, with segmentation

Figure 4.7: Average elapsed time per cell during the exhaustive search of epistasis, for an increasing number of individuals, a fixed number of 680 SNPs, a fixed combination size of three and for both the GCC and Intel compilers.

Table 4.4: Elapsed time, in seconds, and speedup of the 512-bit explicit vector implementation compared to *MPI3SNP*. Results are the average of three executions of a single-threaded third-order search.

		GCC Compiler			Intel Compiler		
SNPs	Inds.	MPI3SNP	Vector	Speedup	MPI3SNP	Vector	Speedup
1000	1000	200.77	39.09	5.14	391.47	40.64	9.63
1000	2000	254.43	50.97	4.99	735.52	48.98	15.02
4000	1000	25 698.86	2527.58	10.17	25 113.33	2626.10	9.56
4000	2000	31 506.60	3401.53	9.26	47 179.35	3288.37	14.35

4.4. Concluding Remarks

As outlined in the previous chapter, the exhaustive search is the only approach that guarantees good epistasis detection powers for all epistasis scenarios studied, with the computational complexity being its main hindrance. In this chapter, we propose different SIMD implementations that exploit the parallelization opportunities inherent to the epistasis detection problem in order to speed up the execution of an exhaustive search. This is achieved by the introduction of AVX Intrinsics during the calculation of the genotype tables, contingency tables and MI metric. We also include general optimization strategies, such as the segmentation of the operation pipeline due to the license-based downclocking on Intel processors, and other optimizations specific to this code, such as the loop unrolling during the calculation of genotype and contingency tables, or the avoidance of logarithms of 0 during the MI calculation. Although this chapter considers a specific exhaustive search algorithm, many of these vectorization and optimization techniques could be directly applied to a multitude of epistasis detection methods in the literature where genotype and contingency tables are constructed to assess the association between a genotype combination and a particular phenotype.

The results obtained highlight the potential of the SIMD parallelization when applied to the epistasis detection problem. For example, the runtime under GCC of an exhaustive search of an interaction consisting of three SNPs from two datasets containing 4000 SNPs and 1000 and 2000 individuals, respectively, was reduced from 428 and 525 minutes using *MPI3SNP* down to 42 and 57 minutes when using the 512-bit vector implementation proposed here.

The autovectorization provided by the compilers showed varying degrees of success attending to the compiler and the operations considered. Intel, for example, was capable of vectorizing all operations while GCC fell short. As for the performance achieved, we observed that optimization flags play a big role in the resulting performance of the code generated. GCC required the `-fast-math` flag to be capable of vectorizing calls to the math library, while Intel improved the performance significantly with the usage of the flag `-qopt-zmm-usage=high`. With respect to performance, Intel's autovectorization remained competitive with the explicit implementations for low-order interaction searches but fell behind when moving past fourth-order interactions. GCC's autovectorization, on the other hand, was never close to the performance of the explicit implementations due to its failure of vectorizing the operations.

Moving forward, with future CPU microarchitectures and the introduction of new AVX extensions, it is reasonable to expect the performance of the SIMD epistasis detection algorithm to improve even further. During the evaluation we saw the effect that the Intel frequency model had on the performance attending to the width of the operations, penalizing the larger vector widths. If these differences in frequency are reduced in upcoming CPUs, the performance will consequently increase. Furthermore, with future AVX instructions, for example, the *popcount* operation from the *AVX512VPOPCNTDQ* extension recently introduced with Intel Ice Lake processors, some operations of the algorithm will allow for a more efficient implementation.

In order to exploit all the computing power that current CPUs and clusters of CPUs offer, instead of the VPU of a single core, the following Chapter 5 presents the combination of the SIMD association test algorithm proposed here with a distributed execution of the exhaustive search for epistasis interactions of any order.

Chapter 5

Any-Order Epistasis Search on CPU Clusters

This chapter presents *Fiuncho*, an exhaustive epistasis detection tool that supports interactions of any given order, and exploits all levels of parallelism available in a homogeneous CPU cluster to accelerate the computation and make it more scalable with the size of the problem. To the best of our knowledge, the proposed implementation is faster than any other state-of-the-art CPU method.

The chapter is organized as follows: Section 5.1 details the parallel epistasis search implemented. Section 5.2 includes the evaluation of *Fiuncho*. At last, Section 5.3 presents the conclusions reached.

5.1. Fiuncho

Fiuncho implements a parallel exhaustive detection method using a static distribution strategy. Given a collection of genotype variants from two groups of samples (cases and controls), the program tests for association every combination of variants for a particular interaction order using the association test presented in Section 1.3, and reports the most associated combinations. To do this, *Fiuncho* combines three different levels of parallelism:

- Task parallelism: the search method is divided into independent tasks that are distributed among the processing resources available in a cluster of CPUs. MPI multiprocessing and multithreading are used for the implementation.
- Data and bit-level parallelism: each task exploits the VPUs by using the SIMD algorithm proposed in Chapter 4. Furthermore, this algorithm uses 64-bit word arrays to represent each of the rows of the genotype tables, and as a consequence of that, each intersection operation (bitwise *AND*) works with 64 samples at once.

This section discusses the method used to exploit the task parallelism. It starts by describing the distribution strategy followed in order to divide and distribute the workload among the computational resources available, and concludes with an algorithm that implements the epistasis search using the presented strategy.

5.1.1. Distribution Strategy

The strategy used to distribute the workload of an epistasis search of any given order follows the same principles of the distribution strategy used in the third-order epistasis search discussed in Section 2.1: divide the workload by combinations of variants of the target interaction size, while reusing as much information as possible through the distribution of smaller combinations. For instance, when searching for fourth-order epistasis, the analysis of the combinations with variants {1,2,3,4}, {1,2,3,5}, {1,2,3,6}, etc. requires the construction of the same genotype table corresponding to the pair {1,2} and the triplet {1,2,3}. Therefore, assigning all these combinations to the same unit will allow the reuse of the genotype tables of {1,2} and {1,2,3} for all fourth-order combinations that contain them.

Fiuncho implements a static distribution strategy in which the combinations of any given order k are distributed among homogeneous computing units using the combinations of size $k-1$, following a RR distribution of the combinations sorted by ascending numerical order. In other words, every combination of size $k-1$ is scheduled among units, and every unit computes all combinations of k variants starting with the given $k-1$ prefix. This strategy finds a middle ground between a good workload balance among computing units and avoiding overlaps in computations among them. By distributing the workload using the $k-1$ combination prefixes we guarantee that every

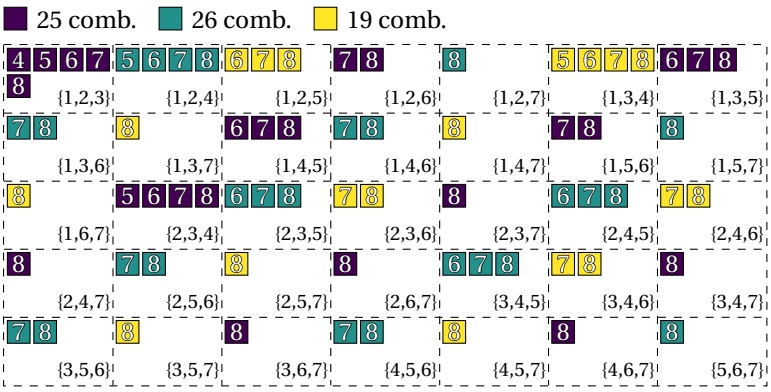


Figure 5.1: Example of the distribution strategy, arranging combinations of four variants among three computing units. Each prefix of three variants (represented as rectangles with dashed lines) is assigned to a unit (shown as different colors) following a RR distribution, and that unit tests for association every combination of four variants starting with the prefix (represented as small colored squares).

combination of size k reuses the genotype tables of its prefix of size $k-1$, but it introduces an overlap among units during the calculation of the tables of the $k-1$ prefix. Nonetheless, repeating these calculations results in a negligible overhead due to the exponential growth of the combinatorial procedure, as the experimental evaluation included in Section 5.2 proves.

Fig. 5.1 exemplifies this strategy, showing the distribution of the computations resulting from a fourth-order search ($k = 4$) of eight variants using three computing units. The figure uses rectangles with dashed lines to represent all prefixes of $k-1 = 3$ variants derived from combining the eight inputs, shown in sorted order from left to right and top to bottom. Each prefix rectangle includes one or more colored squared in its interior, representing a combination of four variants to be tested for association, and the colors indicate the unit which will carry out its test. Every combination under the same prefix is assigned to the same unit, guaranteeing that the genotype table of the prefix is computed only once, and every prefix is assigned to one of the three units following a RR distribution. At the same time, there are small overlaps among the computations corresponding to the different prefixes. For example, the prefix {1,2,3} and {1,2,4} require constructing the same genotype table for the combination {1,2}, and since they were assigned to different units, the table will be constructed more than once. This strategy assigns twenty-five, twenty-six and nineteen combinations to the three com-

puting units, respectively. Although it does not create the most balanced distribution possible, the strategy does not require synchronization or communication between units, takes the reuse of genotype tables into account and achieves very good results for a more realistic input size.

5.1.2. Algorithmic Implementation

With the previous distribution strategy in mind, Algorithm 5.1 presents the pseudocode for the parallel epistasis detection method. It follows the SPMD paradigm in which all computing units execute the same function, while each unit analyzes a different set of variant combinations. The implementation combines MPI multiprocessing with multithreading to efficiently exploit the computational capabilities of CPU clusters. Every MPI process reads the input variants and stores each one in a genotype table, keeping the individual variant information replicated in each process. After that, each MPI process spawns a number of threads that execute the function presented over a different set of variant combinations. The input data is provided to the different threads through shared memory, making an efficient use of the memory inside each node. This procedure allows the parallel strategy to be abstracted from the topology of the cluster, so that the workload is assigned to each core partaking in the computation regardless of its location.

The input arguments to the function are the array a of n genotype tables representing the individual variants, the list of variant combinations l to analyze and the size b of the blocks in which the integer and floating-point vector operations will be segmented. The list of combinations l passed to the function is provided as an iterator that traverses through the combinations assigned to each core without the need of storing the list in memory. In turn, it returns the list of combinations of k variants with the highest MI values.

The algorithm uses the same vector functions of the MI association test presented in Section 4.1: `combine`, `combine_and_popcount` and `MI`. These functions run at very different frequencies in Intel processors [91], measured during the evaluation in Section 4.3, and thus the algorithm divides the operations into different blocks so that each block can operate at a different frequency. This is the same strategy used in Algorithm 4.2.

Algorithm 5.1: `parallel_search`: Distributed algorithm implementing an exhaustive search of any-order epistasis

```
   Procedure sorted_insert(list, ct, mi)
1      if size of list < s or mi > list[0] then
2          Find first i so that list[i] > mi
3          Insert {ct, mi} before i
4          if size of list > s then
5              Remove list[0]
6      return

   Algorithm parallel_search(a, l, b)
           Input:
               a: Array of genotype tables representing n input variants
               l: List of variant combinations to analyze
               b: Size of the block of operations
           Output:
               List of the s highest ranking k-combinations
7      list ← Empty list
8      ct ← Array of b contingency tables of size k
9      cnt ← 0
10     for {i_1, ..., i_{k-1}} in l do
11         gt ← a[i_1]
12         for j ← 2 to k − 1 do
13             gt ← combine(gt, a[i_j])
14         for j ← i_{k-1} + 1 to n do
15             if cnt = b then
16                 for c ← 0 to cnt do
17                     v ← MI(ct[c])
18                     sorted_insert(list, ct[c], v)
19                 cnt ← 0
20             ct[cnt] ← combine_and_popcount(gt, a[j])
21             cnt ← cnt + 1
22     for c ← 0 to cnt do
23         v ← MI(ct[c])
24         sorted_insert(list, ct[c], v)
25     return list
```

Algorithm 5.1 primarily consists of a *for* loop that traverses the list of variant combinations provided to the function (Line 10). The loop begins by computing the genotype table for each combination prefix $\{i_1, \ldots, i_{k-1}\}$. This is done in a progressive manner, starting with the table of the first variant of the prefix i_1, and adding one extra variant to the genotype table at a time using the function `combine`, until the whole prefix is included in the table (Lines 11–13). Once this table is computed, every combination of k variants starting with the given prefix, $\{i_1, \ldots, i_{k-1}, j\}$ with $j \in [i_{k-1} + 1, n - 1]$, is

examined using a *for* loop (Line 14). On each iteration, the genotype frequencies of the combination are obtained through the function `combine_and_popcount`, using the genotype table of the prefix, *gt*, and the table of the selected variant, $a[j]$ (Line 20). The frequencies are stored in an array *ct* of contingency tables. Only when *b* tables have been calculated, the loop enters an *if* branch where the table array *ct* is processed altogether using a *for* loop (Lines 15–19), effectively separating the floating-point vector computations of the `MI` function from the genotype and contingency tables construction operations. On each iteration, a contingency table is processed by computing the MI of the table, and its result is stored in a list of *s* elements, sorted by its MI value using the auxiliary function defined in Lines 1–6.

When the outermost *for* loop ends, the remaining contingency tables stored in the array *ct* are processed (Lines 22–24) and the algorithm returns the sorted list of the top-ranking *s* combinations (Line 25).

The beginning and the end are the only two points in the program requiring synchronization among threads and MPI processes. Once all threads of a process terminate, the different lists of top-ranking combinations kept in the shared memory of the process are joined into one, then sorted by their MI value and truncated to *s* combinations. Analogously, once all MPI processes have assembled their joint lists, the results are gathered into a single joint list through the MPI collective `MPI_Gatherv`. This list is then sorted by MI and truncated to *s* combinations again. To conclude, the program writes the final list to a file and exits.

5.2. Evaluation

This evaluation examines the proposed parallel method in terms of the balance achieved by the parallel distribution, the overhead introduced by the overlap in operations among the different computing units, the parallel efficiency achieved for an increasing number of computing units and a comparison with state of the art exhaustive epistasis detection software. For this evaluation we also use the *SCAYLE* supercomputer, described in Table 4.2. However, in terms of software, we use a more recent version of GCC (11.2) and *glibc* (2.34). Additionally, we use *OpenMPI* version 4.1.

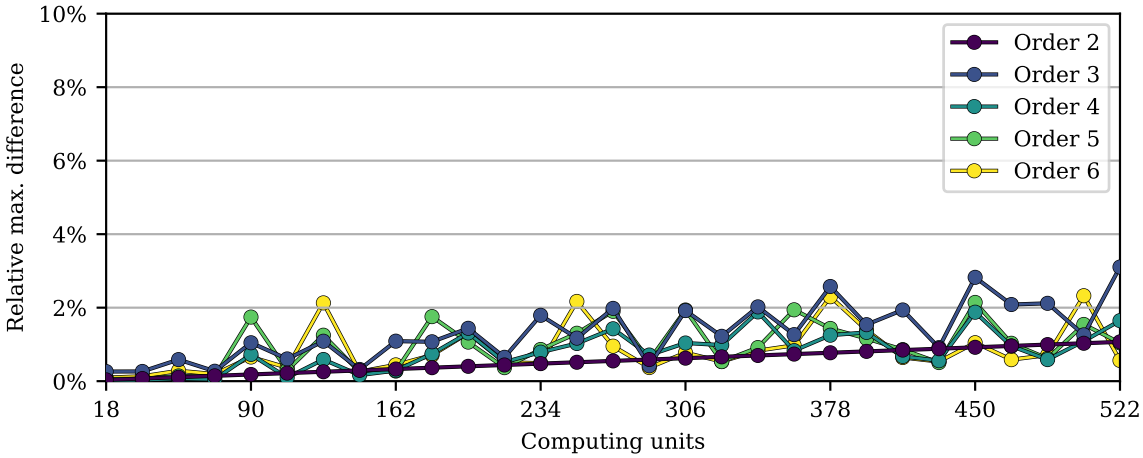


Figure 5.2: Maximum difference of assigned combinations to any computing unit from the average number of combinations assigned per unit, relative to the latter, for orders ranging from 2 to 6.

5.2.1. Parallel Distribution Balance

The distribution strategy presented in Section 5.1.1 does not assign the same exact number of combinations of k variants to test for association to every computing unit. Instead, the strategy makes a compromise between the balance in combinations assigned and the reuse of intermediate results.

In order to evaluate how good the designed strategy is, Fig. 5.2 plots the maximum percentual difference between the number of combinations assigned to a computing unit and the mean number of combinations assigned to any unit, relative to the latter. It can be defined as:

$$100\,\frac{\max d_i - \binom{n}{k}/p}{\binom{n}{k}/p} \tag{5.1}$$

with d_i being the number of combinations assigned to the unit i, n the number of variants, k the size of the combinations and p the number of computing units used. The figure represents the differences in workload distribution using combination sizes from 2 to 6 and a number of units from 18 to 522. In order to keep a similar number of k-combinations, and thus a similar distribution difficulty across combination orders, a number of variants of 48828, 1928, 413, 172 and 100 were used for orders 2 to 6,

Table 5.1: Overhead of the parallel algorithm (run using a single CPU core) compared to a sequential implementation of the same operation, for interaction orders between four and six.

Order	Variants	Combinations	T (s)	T_{alt}(s)	Overhead (%)
4	464	1 906 472 876	1514.61	1526.48	−0.78
5	152	632 671 880	1477.57	1453.21	1.68
6	76	218 618 940	1518.63	1506.17	0.83

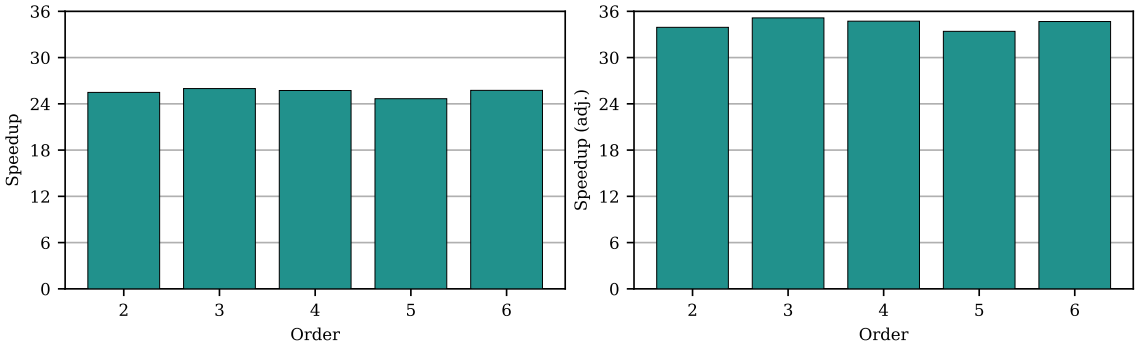
respectively.

The results show that the proposed distribution keeps the differences under 3 % for every scenario tested. For scenarios with larger variant counts, as is the case during the experimental evaluations of Sections 5.2.3 and 5.2.4, the differences in assigned workload are expected to be even smaller.

5.2.2. Parallel Overhead

Although the distribution strategy takes into consideration the reuse of genotype tables to avoid repeating the same operations in different computing units, it certainly does repeat some operations during the construction of those corresponding to the combination prefix assigned by the distribution. In order to measure the overhead introduced, we compared the elapsed time of a single-thread execution of the distributed algorithm with an alternative implementation following the sequential algorithm presented in the previous chapter (Algorithm 4.2), which avoids the repetition of any calculation pertaining to any genotype table.

Table 5.1 represents the overhead, measured as a percentage and calculated as $100 \cdot (T - T_{alt})/T_{alt}$, with T being the elapsed time of the proposed implementation and T_{alt} being the elapsed time of the alternative sequential implementation. The figure omits the second and third-order overheads because, for those combination sizes, the distribution strategy does not produce any overlap in the computations associated with the calculation of genotype tables. The results show that there is no significant difference between the two elapsed times.



(a) Observed speedups (b) Freq. adjusted speedups

Figure 5.3: Speedups of *Fiuncho* for multithread executions using 36 threads, compared to a single-thread run, representing both the observed and frequency-adjusted speedups.

Table 5.2: Elapsed times of single-thread executions of *Fiuncho* for interaction orders between two and six.

ORDER	VARIANTS	COMBINATIONS	ELAPSED TIME (S)
2	184865	17087441680	2491.07
3	3246	5694987980	1612.47
4	464	1906472876	1514.61
5	152	632671880	1477.57
6	76	218618940	1518.63

5.2.3. Speedup and Efficiency

This subsection evaluates the speedup and efficiency of *Fiuncho* using one and multiple nodes. For both scenarios we selected a number of input variants inversely proportional to the order of the interactions so that the elapsed times of the analysis are similar, while the number of samples per variant was kept constant at 2048.

Fig. 5.3 represents the speedups obtained by Fiuncho using a whole node (36 cores) when compared to the elapsed times of single-thread executions shown in Table 5.2, for epistasis orders ranging between two and six. Comparing the speedups from single-thread and multithread executions on an Intel 6240 CPU suffers from the same frequency disparity problem mentioned in the evaluation in Section 3.2 [91]. To represent the efficiency achieved more accurately, the observed and frequency-adjusted speedups are used again. The observed speedup is calculated as T_1 / T_N, with T_1 be-

Table 5.3: Elapsed times of single-node (36 cores) executions of *Fiuncho* for interaction orders between two and six.

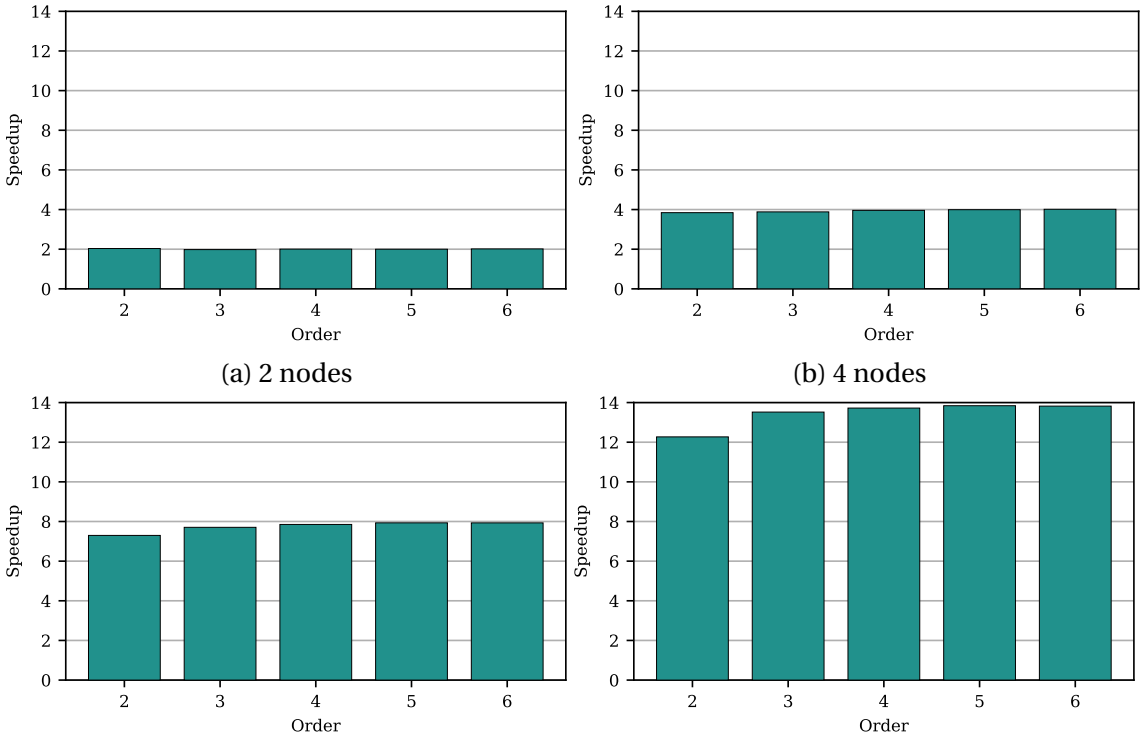
ORDER	VARIANTS	COMBINATIONS	ELAPSED TIME (S)
2	3 755 572	7 052 158 645 806	56 261.70
3	28 576	3 888 727 096 800	42 539.30
4	2409	1 399 760 565 126	43 176.00
5	561	454 852 770 372	42 594.70
6	223	159 602 946 217	43 103.40

ing the elapsed time using a single CPU core and T_N the elapsed time using N CPU cores, and the adjusted speedup is calculated as $T_1/T_N \cdot F_1/F_N$, where F_1 is the average single-core frequency when Fiuncho uses a single core and F_N is the average multicore frequency when N cores are used. The results for a single-node (36 cores) execution show very good efficiencies when the speedup is adjusted for the frequency differences between single-core and multicore executions.

Fig. 5.4 shows the speedups obtained for multinode executions using one MPI process per node with 36 threads each, comparing the elapsed times obtained with a single-node run (36 cores) presented in Table 5.3. The datasets used in this second scenario are substantially larger than those from Table 5.2, in order to keep the elapsed times over an hour long when 14 nodes (504 cores) are used. Here, in a multinode environment, there is no difference between the average CPU frequency of the different nodes since all of them use all the available CPU cores, and thus there is no need to include an adjusted measure of the speedup. Again, results show very good efficiencies except for the second-order interaction. This is due to the large input data for this interaction order, sizing over 29 386 MB and read sequentially, thus limiting the maximum speedup achievable.

5.2.4. Comparison with Other Software

Lastly, the performance of *Fiuncho* was compared with other exhaustive epistasis detection tools from the literature: *MPI3SNP* [2], *MDR* [16] and *BitEpi* [92]. To do this, we compared the elapsed times of all programs when looking for epistasis interactions of orders ranging from two to four. In order to keep the elapsed time constrained, multiple datasets were used containing a number of variants inversely proportional to



(a) 2 nodes (b) 4 nodes

(c) 8 nodes (d) 14 nodes

Figure 5.4: Speedups of *Fiuncho* using 2, 4, 8 and 14 nodes with 36 threads per node, compared to a single-node execution.

Table 5.4: Elapsed time, in seconds, to complete an epistasis search both with *MPI3SNP* and *Fiuncho*, using a different number of nodes and CPU cores.

Order	Variants	Combinations	Nodes	Cores	Fiuncho (s)	MPI3SNP (s)
3	3246	5 694 987 980	1	1	1612.47	12 868.42
3	8505	102 498 733 260	1	18	2240.87	15 312.84
3	10 716	205 033 710 860	1	36	2269.89	16 046.58
3	13 501	410 062 497 750	2	72	2260.79	16 085.50
3	17 010	820 134 519 120	4	144	2291.31	16 186.12

the order of the epistasis search. The number of samples per variant, however, is 2048 for all datasets. Since *MDR* is considerably slower than the rest of programs, smaller data sizes were used for its evaluation.

Table 5.4 compares the elapsed times of *Fiuncho* and *MPI3SNP*, the tool previously developed by us and described in Chapter 2. This program is limited to third-order searches, thus the evaluation only considers this interaction order. It implements MPI

Table 5.5: Elapsed time, in seconds, to complete an epistasis search both with *BitEpi* and *Fiuncho*, using a different number of threads and orders. The total workload between orders was kept as similar as possible.

ORDER	VARIANTS	COMBINATIONS	CORES	FIUNCHO (S)	BITEPI (S)
2	184 865	17 087 441 680	1	2491.07	18 090.91
2	784 314	307 573 833 141	18	3582.01	22 294.39
2	1 109 187	615 147 345 891	36	3797.39	23 365.74
3	3246	5 694 987 980	1	1612.47	5417.15
3	8505	102 498 733 260	18	2240.87	7474.48
3	10 716	205 033 710 860	36	2269.89	7564.48
4	464	1 906 472 876	1	1514.61	2202.65
4	954	34 296 318 126	18	2101.60	3239.25
4	1134	68 539 472 001	36	2111.74	3246.72

multiprocessing, so different scenarios were considered which include single-thread, single-node and multinode configurations. Both *MPI3SNP* and *Fiuncho* assign one MPI process per node, and create as many threads per process as cores available in each node. The results show that *Fiuncho* is significantly faster than *MPI3SNP* in all the evaluated scenarios.

Table 5.5 compares the results of *BitEpi* with *Fiuncho*. *BitEpi* is a very novel program that only supports interaction orders between two and four, thus the evaluation is restricted to those orders. Additionally, *BitEpi* supports multithreading, therefore single-thread and multithread scenarios are used. *BitEpi* uses a substantially different association test with a time-complexity of $O(km)$, while the association test used in *Fiuncho* has a time-complexity of $O(3^k m)$. This can be observed in the results as a shrinking difference between the elapsed times with the epistasis size. Despite this, *Fiuncho* is still faster in all configurations tested. Furthermore, *BitEpi* does not support multinode environments and can only exploit the hardware resources of a single node, while *Fiuncho* can use as many resources as available in order to reduce even further the elapsed time of the search.

Lastly, Table 5.6 compares the elapsed time of *MDR* with *Fiuncho*, using a more limited number of variants than previous comparisons. *MDR* is a relatively old program written in Java, but we decided to include it due to its relevance in the field. It implements an epistasis search supporting interactions of any order, although its elapsed time quickly becomes prohibitive even with a reduced input size, so we decided to

Table 5.6: Elapsed time, in seconds, to complete an epistasis search both with *MDR* and *Fiuncho*, using a different number of threads and orders. The total workload between orders was kept as similar as possible.

ORDER	VARIANTS	COMBINATIONS	CORES	FIUNCHO (S)	MDR (S)
2	9300	43240350	1	6.26	3571.77
2	39455	778328785	18	13.16	8656.46
2	55797	1556624706	36	16.04	10285.10
3	580	32350660	1	9.22	3204.88
3	1518	581842316	18	12.94	4710.82
3	1913	1164963436	36	13.34	6598.97
4	160	26294360	1	20.97	3767.28
4	328	473490550	18	29.04	4710.03
4	390	949173615	36	29.37	6491.59

keep the interaction orders between two and four. *MDR* supports multithreading, so single-thread and multithread scenarios were considered in this evaluation. Results show a massive difference in elapsed times, with an average speedup of 358 of *Fiuncho* over *MDR*. This speedup could be increased even further if we considered multinode scenarios for larger datasets, something that *MDR* does not support unlike *Fiuncho*.

5.3. Concluding Remarks

This chapter presents *Fiuncho*, an epistasis detection program for any-order epistasis detection. It is the culmination of this thesis, combining the SIMD implementation of the association test for the *x86_64* architecture presented in Chapter 4, with a distributed search algorithm that supports interactions of any order, inspired by the *MPI3SNP* method described in Chapter 2.

Fiuncho shows an exceptional performance, with a parallel strategy that balances the workload remarkably well, obtaining computational efficiencies close to an ideal growth with the hardware resources provided. When compared to existing epistasis detection software, *Fiuncho* offers support for a wider scope of application with no limit on the target epistasis size, and performs the fastest of all programs considered in this study. For example, on average, *Fiuncho* is seven times faster than its predecessor, *MPI3SNP* [2], three times faster than *BitEpi* [92] and 358 times faster than *MDR* [16]. Moreover, the speedups over *BitEpi* and *MDR* could be multiplied if larger experiments

on multinode environments were considered, as they are restricted to the hardware resources available in a single node.

Chapter 6

Calculating Penetrance Tables of High-Order Epistasis Models

Simulation is essential in order to study and develop new algorithms or methods for epistasis detection. Simulations offer a controlled environment for testing the accuracy of new methods where the expected results are known beforehand. In contrast, real world data are more costly to acquire and provide no direct way of knowing which result is correct. Here, we introduce a software library named *Toxo*, capable of simulating penetrance tables that can be used later by epistasis simulators to generate data. Thanks to it, the review presented in Chapter 3 includes datasets with epistasis interactions following a particular epistasis model.

The chapter is organized as follows: Section 6.1 defines what a penetrance table is, and the different strategies that are used in the literature in order to obtain them. Section 6.2 describes the general procedure that *Toxo* follows in order to calculate penetrance tables. Section 6.3 details how the library was implemented and how can it be used in conjunction with existing simulators. Section 6.4 evaluates Toxo in terms of runtime and precision of the results, compares the ability to obtain penetrance tables with other state-of-the-art applications and discusses what Toxo can and cannot do. At last, Section 6.5 highlights the conclusions reached and some future areas of improvement.

6.1. Penetrance Tables

Penetrance tables define the different probabilities of expressing the phenotype considering the genotype that each individual possess. They are the most common way to characterize an epistatic relationship, and they are commonly used by simulation software in order to define epistasis interactions. Nonetheless, not all simulators implement the logic necessary to generate the penetrance tables themselves. *SimuPOP* [93], *HapSample* [94], or *SBVB* [95], for example, can simulate synthetic datasets employing penetrance tables, but they cannot create them.

Three general approaches are used to create the penetrance tables. The first and most simple approach consists in using an epistasis model. Epistasis models are mathematical relationships that define the penetrance value for each genotype combination as a function of one or more variables, each one usually representing a statistical parameter of the interaction. Fig. 6.1 includes some examples of epistasis models. Marchini et al. in [84], for example, use two different parameters to define their models: the baseline effect (α), the genetic effect present at every locus independently of the actual allele combination, and the genotypic effect (θ), the increase in the odds of the disease beyond the baseline level due to genetic interaction. From these models, a penetrance table can be obtained by giving values to every parameter. However, since penetrances are probability values, they can only be inside the interval $[0,1]$ and, therefore, there are some restrictions on how the parameter values can be combined. An example of the usage of epistasis models to generate penetrance tables as described can be found in [96].

The second approach is to impose a set of characteristics that should be fulfilled by the simulated population under study and find a penetrance table that complies with these requirements. Parameters model certain characteristics of the population, and the most common are the prevalence $P(D)$ (representing the proportion of individuals in a population carrying the phenotype of study) and the heritability h^2 (representing the amount of phenotypic variation that corresponds to genetic variation). Finding a table with such requirements is a more complex process than using an epistatic model, therefore a software tool is needed. In this regard, *GAMETES* [83] is an epistasis simulation software that uses a stochastic method to find a penetrance table with the desired prevalence and heritability levels. It is also able to generate population samples from

	BB	Bb	bb
AA	α	$\alpha(1+\theta)$	$\alpha(1+\theta)^2$
Aa	$\alpha(1+\theta)$	$\alpha(1+\theta)^2$	$\alpha(1+\theta)^3$
aa	$\alpha(1+\theta)^2$	$\alpha(1+\theta)^3$	$\alpha(1+\theta)^4$

(a) Marchini et al.'s additive model [84]

	BB	Bb	bb
AA	α	α	α
Aa	α	$\alpha(1+\theta)$	$\alpha(1+\theta)^2$
aa	α	$\alpha(1+\theta)^2$	$\alpha(1+\theta)^4$

(b) Marchini et al.'s multiplicative model [84]

	BB	Bb	bb
AA	α	α	α
Aa	α	$\alpha(1+\theta)$	$\alpha(1+\theta)$
aa	α	$\alpha(1+\theta)$	$\alpha(1+\theta)$

(c) Marchini et al.'s threshold model [84]

	BB	Bb	bb
AA	α	α	α
Aa	αf	α/f	α/f
aa	αf	α/f	α/f

(d) Shang et al.'s third model [50]

Figure 6.1: Examples of penetrance table models.

these tables. *GenomeSIMLA* [97] is another simulator capable of finding a penetrance table under prevalence and heritability constraints. In this case, it uses a GA to reach a solution.

The third and last approach consists in combining the two previous methods: the use of epistasis models together with a set of parametric restrictions. This approach has the advantage of modeling the interaction using the model variables, while also modeling some population characteristics using the parametric restrictions. Consequently, finding a penetrance table is a significantly more complex task. *EpiSIM* [98] and *gs* [99] are simulators that fall into this hybrid approach. *gs* offers the ability to create penetrance tables for nine embedded second-order models, based on the genotype odds ratio(s) for each locus and the prevalence of the desired phenotype. The usability of *gs* is especially limited due to its restricted set of models. *EpiSIM*, on the other hand, can create penetrance tables of up to fourth-order and simulate population samples from them. It allows us to specify penetrance values as a function of two variables (i.e., it uses *bivariate* penetrance functions) and it also permits specifying the desired values of prevalence and heritability. The *EpiSIM* implementation attempts to find a value for the model variables by solving the equation system made of the prevalence and heritability equations, respectively defined as:

$$P(D) = \sum_i P(D|g_i)P(g_i) \tag{6.1}$$

$$h^2 = \frac{\sum_i \big(P(D|g_i) - P(D)\big)^2 P(g_i)}{P(D)\big(1 - P(D)\big)} \tag{6.2}$$

where $P(D|g_i) = f_i(x, y)$ is the proportion of individuals showing trait D when having the genotype g_i, $P(g_i)$ is the population frequency of the genotype g_i and $f_i(x, y)$ is the function of two variables that is defined by the epistasis model. *EpiSIM* seeks to find the penetrance table or tables that meet certain prevalence and heritability constraints by solving an equation system made of the two previous expressions. This results in a system with two equations and two unknowns: the two variables of the epistasis model.

Although this approach can work for second-order models and low prevalence and heritability values, *EpiSIM* struggles to find solutions to higher-order models or more realistic parameter values. *Toxo*, the software library presented, can calculate penetrance tables from models containing bivariate penetrance functions with no limitation on the interaction order. *Toxo* allows the user to create penetrance tables for a specified epistasis model maximizing the prevalence or heritability when one of the two is constrained. These tables can be used by other simulation packages to generate the dataset with the embedded epistasis model and the parametric restriction specified.

6.2. Method

Based on the limitations described in the previous section, Toxo implements a method to circumvent these issues and enable simulators to generate high-order epistasis in their data.

6.2.1. An Alternative Approach to Calculating Penetrance Tables

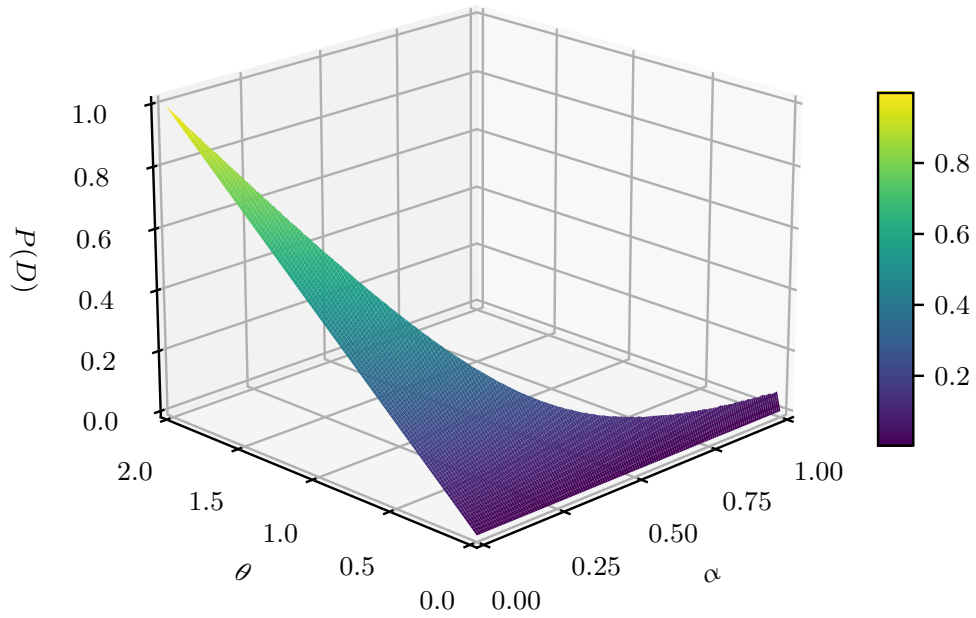
One of the biggest hurdles of solving the equation system used in *EpiSIM* (a system of equations composed of Equations 6.1 and 6.2) is the formulation of incompatible equation systems by requesting a combination of prevalence and heritability values that cannot be achieved by the model introduced. If we take the additive model as an example, shown in Fig. 6.1a, the prevalence and heritability can be expressed as functions of the model parameters α and θ, and a prevalence and heritability value can be obtained for any combination of α and θ. However, not every combination of values represents a valid result. Penetrance values are probabilities, and they are restricted to the interval $[0,1]$. In other words, the epistasis model imposes a restriction on which combinations of α and θ are valid, and thus limits the values of prevalence and heritability that can be simultaneously achieved by a model. Fig. 6.2 represents the prevalence and heritability as functions of two variables for the second-order additive model, giving a visual representation of this phenomenon. The two subfigures show no common point (α,θ) to both graphs where $P(D) = 0.8$ and $h^2 = 0.2$, proving it is not possible to reach both these values simultaneously using this model.

To overcome this limitation, instead of finding a specific combination, *Toxo* maximizes one of the two parameters (prevalence or heritability) when the other is fixed. Once the maximum is calculated, the interval of achievable values is perfectly defined as the interval between 0 and the maximum. Following this approach, the likelihood of formulating an incompatible system when no information of the model is known is significantly reduced, since most of the models achieve all prevalences and heritabilities individually at some point. *Toxo* also considers the valid range of penetrance values as constraints to the equation system to be solved. Depending on the parameter to maximize (prevalence or heritability) the method slightly varies, so both will be explained in detail.

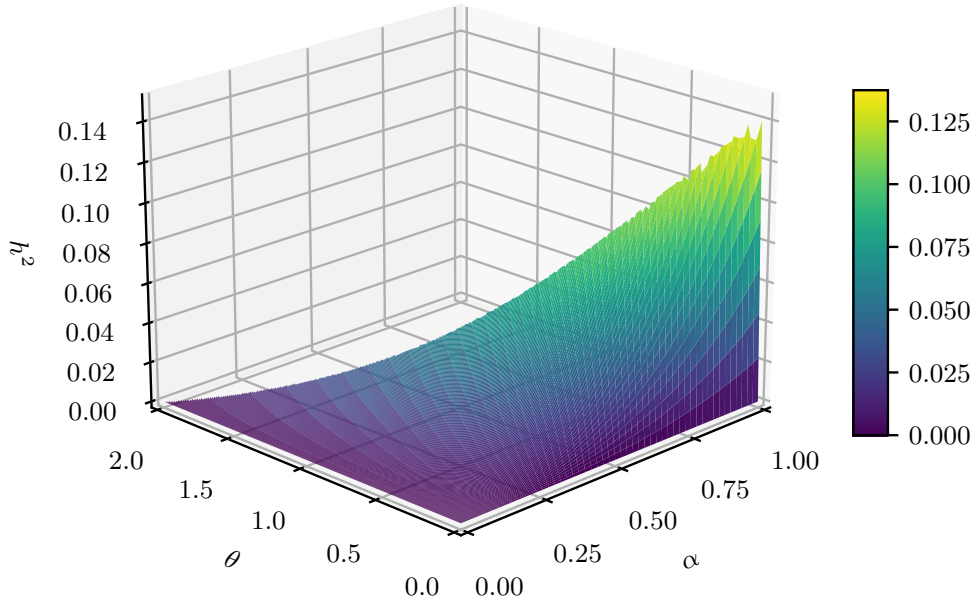
Taking into account Equation 6.1, maximizing the prevalence means maximizing the sum:

$$\sum_i \big(P(D|g_i)P(g_i)\big) \tag{6.3}$$

where $P(D|g_i)$ is a function of the model variables (generally referred to as x and y)



(a) Prevalence function



(b) Heritability function

Figure 6.2: Prevalence and heritability as functions of α and θ for the second-order additive model shown in Fig. 6.1a, using MAF = 0.25, $\alpha \in [0,1]$ and $\theta \in [0,2]$. Note that prevalence values closer to 0 and heritability values higher than 0.15 can be achieved for values of θ higher than two, outside the area represented in the figure.

and $P(g_i)$ is constant for fixed MAFs, assuming Hardy-Weinberg equilibrium between the three genotypes at each locus and linkage equilibrium among the loci [100, 98]. To simplify the maximization process, we impose two restrictions to the input model:

1. All model expressions must be monotonically non-decreasing when x and y are real positive numbers.
2. The penetrance expressions must be sortable when x and y are real positive numbers.

These restrictions include the vast majority of models used in the literature, as will be discussed in Section 6.4.2.

If the penetrance expressions are monotonically non-decreasing and sortable, all expressions will increment proportionally when increasing their variables. Consequently, their sum will reach its maximum value when the largest $P(D|g_i)$ expression also takes its maximum. Since penetrances are probabilities, their maximum value is 1. Therefore, we can obtain the maximum prevalence for a model, given a heritability value, by solving an equation system made of this heritability constraint and the condition of maximum prevalence:

$$\begin{aligned} \frac{\sum_i \big(P(D|g_i) - P(D)\big)^2 P(g_i)}{P(D)\big(1-P(D)\big)} &= h^2 \\ max\big(P(D|g_i)\big) &= 1 \end{aligned} \tag{6.4}$$

The last step is to discard any solution with negative values for any of the variables of the model. The restrictions on the models are only true for real positive numbers and, as a result, there is no guarantee that negative solutions represent a maximum on the model.

An analogous process is followed to maximize the heritability when fixing the prevalence. On the heritability expression (Equation 6.2), the only variable term is the sum in the numerator, since the prevalence and MAFs are fixed. Therefore, to maximize it we need to maximize the sum:

$$\sum_i \big(P(D|g_i) - P(D)\big)P(g_i) \tag{6.5}$$

Using the same two restrictions as before, the sum will be maximum when the largest penetrance expression takes its maximum value since all expressions are monotonically non-decreasing. Again, we can obtain the maximum heritability for a model given its prevalence value by solving an equation system made of the prevalence expression and the condition of maximum heritability:

$$\begin{aligned} \sum_i \big(P(D|g_i)P(g_i)\big) &= P(D) \\ max\big(P(D|g_i)\big) &= 1 \end{aligned} \tag{6.6}$$

6.2.2. Numerical Example

Assume that we work with the second-order additive model shown in Fig. 6.1a. Our objective is to maximize the prevalence for a fixed MAF and heritability (in this example, 0.25 and 0.2, respectively). The first step consists in verifying that the model meets the restrictions:

- Non-decreasing monotone expressions in the real positive number space: model expressions are monotonic in the real positive number space when its partial derivatives show no change in the sign for $x > 0$ and $y > 0$. The partial derivatives of all polynomial expressions for the second-order model are:

$$\begin{aligned}
\frac{\partial}{\partial x}(x) &= 1 \\
\frac{\partial}{\partial y}(x) &= 0 \\
\frac{\partial}{\partial x}\big(x(1+y)\big) &= 1+y \\
\frac{\partial}{\partial y}\big(x(1+y)\big) &= x \\
\frac{\partial}{\partial x}\big(x(1+y)^2\big) &= (1+y)^2 \\
\frac{\partial}{\partial y}\big(x(1+y)^2\big) &= x(2y+2) \\
\frac{\partial}{\partial x}\big(x(1+y)^3\big) &= (1+y)^3 \\
\frac{\partial}{\partial y}\big(x(1+y)^3\big) &= x(3y^2+6y+3) \\
\frac{\partial}{\partial x}\big(x(1+y)^4\big) &= (1+y)^4 \\
\frac{\partial}{\partial y}\big(x(1+y)^4\big) &= x(4y^3+12y^2+12y+4)
\end{aligned} \tag{6.7}$$

All these derivatives are positive when $x > 0$ and $y > 0$.

- Sortable expressions in the real positive number space: all polynomial expressions can be sorted unequivocally:

$$x \le x(1+y) \le x(1+y)^2 \le x(1+y)^3 \le x(1+y)^4,\ \forall x, y \in \mathbb{R},\ x, y \ge 0 \tag{6.8}$$

After verifying that the model is appropriate for this method, the next step is to calculate the probability associated with each combination of two genotypes. Assuming linkage equilibrium between the two loci, and under the Hardy-Weinberg principle, the probability of a genotype can be calculated as the product of the probabilities of each allele [100]. This can be extended to any order of interaction by including the probabilities of each intervening allele in the product, provided that the same assumptions hold true. Thus, for an associated MAF of 0.25 for the two loci, the probabilities of each allele are $p = 1/4$ and $q = 1 - p = 3/4$, and the resulting allele combination probabilities are those shown in Table 6.1.

Table 6.1: Genotype probabilities of two loci combinations with the same MAF = 0.25.

AABB	AABb	AAbb	AaBB	AaBb	Aabb	aaBB	aaBb	aabb
81/256	27/128	9/256	27/128	9/64	3/128	9/256	3/128	1/256

Table 6.2: Penetrance table of a second-order additive model with MAF = 0.25, $h^2 = 0.2$ and maximum $P(D)$.

AABB	AABb	AAbb	AaBB	AaBb	Aabb	aaBB	aaBb	aabb
0.0019	0.0092	0.0439	0.0092	0.0439	0.2096	0.0439	0.2096	1.0000

Equations 6.4 have to be used in order to find the maximum prevalence for a fixed heritability value. The resulting equation system after replacing $P(D|g_i)$ with the model expressions from Fig. 6.1a, and $max(P(D|g_i))$ with the maximum expression, $x(1+y)^4$, is:

$$\begin{aligned} \frac{3xy^2\left(85y^6+672y^5+3264y^4+9728y^3+19968y^2+24576y+16384\right)}{(y+4)^4\left(256-xy^4-16xy^3-96xy^2-256xy-256x\right)} &= 0.2 \\ x(1+y)^4 &= 1 \end{aligned} \tag{6.9}$$

The solution to the system, for $x \geq 0$ and $y \geq 0$, is $x = 0.0019$ and $y = 3.7714$. Table 6.2 shows the resulting penetrance table, which has an associated prevalence and heritability of 0.0275 and 0.2 respectively.

6.3. Toxo

This section covers the software architecture of the *Toxo* library, how it is intended to be used and concludes with a complete usage example.

6.3.1. Overview

Toxo is a MATLAB library designed for calculating penetrance tables using epistasis models containing bivariate penetrance functions, maximizing the prevalence or

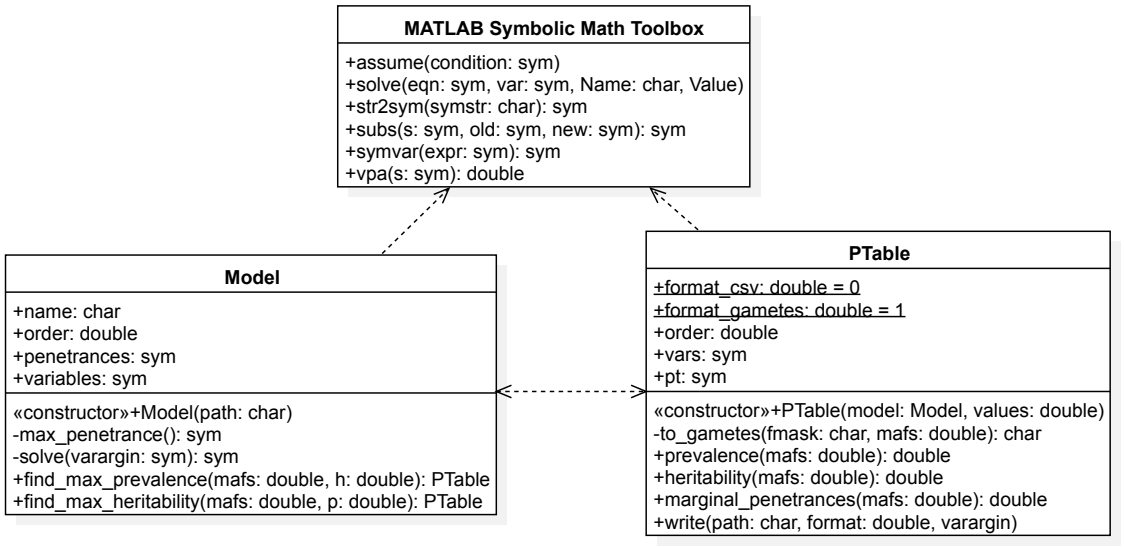


Figure 6.3: Class diagram of Toxo, representing its two classes `Model` and `PTable`, as well as all their attributes and methods. Class `Model` represents an epistasis model containing bivariate penetrance expressions and offers methods for calculating penetrance tables according to its definition. Class `PTable` represents a penetrance table and offers methods for calculating parameters from the table and writing it into a file.

heritability when one of the two is set. It finds the combination of the two variables from the model that results in a penetrance table where the prevalence is maximum if the heritability was constrained, or the heritability is maximum if the prevalence was the constraint. *Toxo* does not generate population samples from the tables; instead, it relies on other programs, such as *GAMETES* [83], to simulate the samples using these tables.

The library consists of two classes, `Model` and `PTable`, which encapsulate all the functionality, as represented in Fig. 6.3. The `Model` class constructor reads the model (provided as a text file) and creates an object representing it. Its instance offers methods for calculating the penetrance table with the maximum heritability for a certain prevalence, or the table with the maximum prevalence for the specified heritability. These methods return instances of `PTable`, representing the calculated penetrance table and offering methods for, among other things, writing the table to a file using different formats. In the event of not finding a penetrance table with the desired characteristics, an exception will be raised.

Toxo uses the Symbolic Math Toolbox of MATLAB [101] to represent the models and to calculate the resulting penetrance table. This allows the user to control the precision of the results by changing the precision on all the operations computed within *Toxo*. If the target prevalence or heritability is a number close to 0 or 1 (the minimum and maximum values, respectively), it may be necessary to increase the number of digits to reduce the error in precision (using the MATLAB function `digits`).

6.3.2. Integration with Other Software

Toxo only calculates penetrance tables and it is intended to be used together with other software to complete the simulation of the data samples whose interactions correspond to those of the considered model. The design of *Toxo* is consequently focused on the integrability with third-party software. To accomplish this, *Toxo* relies on text files to communicate with other programs.

An example of this integration is included with the source code of the tool[1]. In this case, *GAMETES* is used to simulate data using the penetrance tables generated by *Toxo*. The models are read by *Toxo* and its outputs (the calculated penetrance tables) are written following the *GAMETES*' format. *GAMETES* then directly reads the file written by *Toxo*, a file comprised of all penetrances for the different allele combinations, and generates population samples using its own simulation method. Once it finishes, the result is a data file which segregates individuals as cases and controls, and for each individual the same genotype markers are specified.

Toxo offers complete flexibility on the output format of the table thanks to its object-oriented implementation, and it can be easily extended to support any other format required by a simulator.

6.3.3. Usage Example

For the simple reason that *Toxo* is a programming library, it does not offer a graphical interface. Instead, it offers an Application Programming Interface (API) to its users so that any of its functions and methods can be used within any script or program. In

[1] https://github.com/UDC-GAC/toxo/blob/master/examples/usage_example.m

order to describe the usage of *Toxo*, this section will exemplify how to generate a penetrance table for the second-order model shown in Fig. 6.1a with MAF = 0.25 for both loci that can be loaded directly into *GAMETES* [83] to generate data samples.

The first step to create a penetrance table is to define the epistasis model to be used. It must be written to a file using CSV (Comma-Separated Values) format, where rows correspond to the different genotypes and two columns define the genotype and its associated penetrance expression. The two variables are arbitrarily named x and y (*Toxo* interprets any alphabetic characters in the penetrance expressions column as variable names). To define the second-order additive model, a file named `model.csv` is created containing the following information:

```
AABB, x
AABb, x*(1+y)
AAbb, x*(1+y)^2
AaBB, x*(1+y)
AaBb, x*(1+y)^2
Aabb, x*(1+y)^3
aaBB, x*(1+y)^2
aaBb, x*(1+y)^3
aabb, x*(1+y)^4
```

Once the model file is created, an instance of the class `Model` can be created by reading it:

```
m = toxo.Model('model.csv')
```

From this `Model` instance, the penetrance table with maximum prevalence can be found using the method `find_max_prevalence`. The parameters of this method are the MAF for each of the two loci of the model (given as a vector) and the heritability constraint. Following the example, the function call to create a penetrance table for the model with MAF$s = 0.25$ and $h^2 = 0.2$ is:

```
pt = m.find_max_prevalence([0.25, 0.25], 0.2)
```

In the case of looking for the table with maximum heritability, the method to be called instead is `find_max_heritability`. The parameters of this method are, again, the MAF for each of the two locus of the model (given as a vector) and the prevalence

constraint of 0.1 instead of the heritability:

```
pt = m.find_max_heritability([0.25, 0.25], 0.1)
```

Finally, the calculated penetrance table can be written to a file so that a simulator can make use of it to generate datasets, which can be done using the method `write` of the class `PTable`. The output format is chosen using different constants statically declared inside the class `PTable`. In our example, to use *GAMETES* we have to introduce the `format_gametes` constant:

```
pt.write('table.txt', toxo.PTable.format_gametes, [0.25, 0.25])
```

The resulting file `table.txt` can be loaded as a model inside *GAMETES*, and data can be simulated from it. The code included in this example is also available at the GitHub repository, which can be executed line by line to further comprehend the usage of *Toxo*[1].

6.4. Evaluation and Discussion

The evaluation of *Toxo* is divided into two parts: the quality of the implementation achieved in terms of the precision of the results and the elapsed time required to obtain them, and the scope of models from the literature supported by *Toxo*. All the tests were run on a 64-bit Linux machine with two eight-core Intel E5-2660 CPUs and 64 GB of RAM, using the command line interface of MATLAB version R2018a.

6.4.1. Precision and Runtime Evaluation

A battery of tests was developed to evaluate the precision of the results (the difference between the requested and the observed heritability) and the runtime. All executions were repeated five times and their runtimes averaged to avoid outliers. Table 6.3 shows the results for the additive, multiplicative and threshold models [84] (represented in Figs. 6.1a, 6.1b and 6.1c), generalized for third and fourth-order, and for a variety of MAF and heritability values. The evaluation is focused on the heritability since

[1] https://github.com/UDC-GAC/toxo/blob/master/examples/usage_example.m

Table 6.3: Precision error of the heritability obtained for the penetrance table and execution time, calculated under several models, MAF and heritability configurations.

MODEL	ORDER	MAF	H^2	ERROR	TIME (S)
Additive	3	0.1	0.1	0	7.06
Additive	3	0.1	0.8	1.31×10^{-5}	7.08
Additive	3	0.4	0.1	0	6.89
Additive	3	0.4	0.8	9.99×10^{-16}	6.95
Additive	4	0.1	0.1	1.58×10^{-12}	14.17
Additive	4	0.1	0.8	4.04×10^{-12}	13.14
Additive	4	0.4	0.1	0	13.59
Additive	4	0.4	0.8	3.92×10^{-3}	13.61
Multiplicative	3	0.1	0.1	0	8.60
Multiplicative	3	0.1	0.8	0	8.51
Multiplicative	3	0.4	0.1	0	8.03
Multiplicative	3	0.4	0.8	0	7.82
Multiplicative	4	0.1	0.1	0	142.32
Multiplicative	4	0.1	0.8	0	145.94
Multiplicative	4	0.4	0.1	0	90.05
Multiplicative	4	0.4	0.8	0	85.42
Threshold	3	0.1	0.1	0	2.55
Threshold	3	0.1	0.8	0	2.54
Threshold	3	0.4	0.1	0	2.50
Threshold	3	0.4	0.8	0	2.50
Threshold	4	0.1	0.1	0	3.57
Threshold	4	0.1	0.8	0	3.57
Threshold	4	0.4	0.1	0	3.59
Threshold	4	0.4	0.8	0	3.58

it is the parameter with the most interest in case-control studies, whereas the prevalence is not as important because having a fixed number of cases and controls negates the effect of phenotype frequency in a non-controlled environment. The selection of models ranges from a very simple model like the threshold (where all the polynomials inside the model are of first degree) to a more complex one like the multiplicative (where the degree is generally higher). The MAF and heritability combinations were also chosen to show a wide spectrum of values. Results show that the precision error is almost nonexistent for every test. As for the runtimes, all the tables were calculated in under a quarter of a minute, with the only exception being the fourth-order multiplicative model, which took a little more than two minutes.

To compare these results with state-of-the-art competitors, the same table configurations were attempted in *EpiSIM* [98]. Although *gs* [99] can also calculate penetrance

tables from epistasis models containing bivariate functions, it is not included in the comparison as it does not allow modifying the second-order embedded models included within the program. *EpiSIM*, on the other hand, requires both the prevalence and heritability to obtain a penetrance table. To make a fair comparison, two different cases were tested for each of the configurations defined: one with the exact same prevalence and heritability combination obtained by *Toxo*, and a second one with the former heritability and a fixed prevalence value (1×10^{-20}), supposedly easier to find since it is below the maximum. Despite this, *EpiSIM* could not find a single table for any of the tests.

6.4.2. Model Restrictions and Existing Epistasis Models

As explained in Section 6.2, *Toxo* only admits models that meet two conditions:

- All model expressions are monotonically non-decreasing when the two model variables take real positive numbers.
- The penetrance expressions are sortable when the two penetrance variables take real positive numbers.

Nevertheless, these two conditions are met by most of the epistasis models that are actively used in the literature. These include Marchini et al.'s second-order models [84] as well as their generalizations to higher orders, the epistasis models proposed in experimental evaluation of *BEAM* [96], and the heterogeneity models introduced by Neuman et al. [100].

The only example that we could find of a bivariate model that does not comply with the required conditions is Shang et al.'s third model [50], also shown in Fig. 6.1d. In this model, the expression α/f is not monotonically increasing since it increases for $f \in [0,1]$ and decreases for $f \in [1,\infty)$. Furthermore, the expressions of the model cannot be sorted for the real positive number space, as α is greater or equal than α/f for $f \in [0,1]$ but lower for $f \in (1,\infty)$.

Recent studies that include simulations based on epistasis models to generate their evaluation data [102, 103, 104] settle on low-order models whose heritability values are

worryingly moderate. However, real-world diseases are usually determined by a higher number of genes [105] and a higher heritability [106, 107]. Our assumption is that previous works needed to use non-realistic low-order models and non-realistic heritability values due to limitations of state-of-the-art simulators, which are incapable of generating synthetic data with high heritability levels for high-order models. *Toxo* can facilitate current studies to overcome this limitation by finding appropriate penetrance tables that allow current simulators to create samples that resemble real-world data more closely.

6.5. Concluding Remarks

This chapter introduced *Toxo*, a MATLAB library capable of calculating penetrance tables from models containing bivariate penetrance functions with no limitations on the interaction order. It allows the user to maximize the prevalence of the resulting table when the heritability is constrained and vice versa. In addition, *Toxo* can be easily integrated with other existing simulators to generate datasets that include the epistasis relationships described in the penetrance table. It was used in Chapter 3 to generate the datasets with marginal effects used to compare the high-order epistasis detection tools available in the literature.

Thanks to the mathematical method used underneath, *Toxo* can calculate penetrance tables with prevalence and heritability values much higher than those observed in the state-of-the-art. The majority of the works in the literature, if not all, use heritabilities under 0.2 for high-order penetrance tables. However, it is believed that real world diseases present higher heritabilities. *Toxo* provides researchers with a library to generate penetrance tables and, in consequence, data samples that resemble characteristics from real world diseases more closely.

Empirical results show that *Toxo* is capable of calculating penetrance tables for high-order models according to the specified parameters with barely any precision error. Third-order tables can be obtained in under 10 seconds, and fourth-order tables in about 2 minutes.

Although *Toxo* solves several of the shortcomings of state-of-the-art simulators, it also has its own limitations. First, MATLAB is a commercial software and the user will

need a license to run *Toxo*. In addition, *Toxo* is a library and thus, it requires certain programming knowledge to use it. It also presents limitations in the accuracy of the results, motivated by the compromise between computing time and precision that users are forced to make in MATLAB when selecting the number of decimals to operate with variable-precision arithmetic. To solve all these issues, recently we developed *PyToxo* [7], a reimplementation of *Toxo* in Python. *PyToxo* provides different user interfaces, including a Graphical User Interface (GUI), for ease of use. It also improves the results achieved by *Toxo* in terms of runtime, scope of application and precision of the results.

Chapter 7

Conclusions and Future Work

In the last decade, GWAS have identified hundreds of genetic variations associated with complex human traits and diseases. However, these variants only explain a small fraction of the observed heritability. High-order epistasis is hypothesized to be one of the contributing factors for this missing heritability. A plethora of methods have been published to study epistasis, although no definitive solution has been found to the problem. Currently, epistasis detection is still an active research field with new approaches published every month. In this context, this thesis makes the following contributions:

- **A parallel CPU/GPU solution for exhaustive third-order epistasis detection.** A new method was developed to exhaustively search epistasis interaction in homogeneous CPU or GPU clusters. The implementation combines MPI with multithreading and CUDA to exploit the parallelism available in clusters consisting of multiple CPUs or GPUs located in different nodes and interconnected through a network. The static distribution implemented does not require communications among computing units outside the initialization and completion of the program, and is ideal for analyzing large datasets using a considerable number of nodes.

- **An extensive survey of high-order epistasis detection methods.** The study includes 27 different methods, categorizes them in six different groups attending to the similarities and differences among them, and presents an evaluation in

terms of elapsed time, epistasis detection power and presence of false positives in the results using a common set of experiments for all methods. The most notable conclusions are the differences in detection power between the presence and absence of marginal effects in the epistatic interactions. Despite the high number of non-exhaustive methods finding success in the identification of the epistatic interactions with marginal effects, only the exhaustive methods can reliably identify the interactions in the absence of these effects. On the other hand, the survey reveals that false-positives are not considered in the design of most methods, with only *BEAM3* being capable of reliably finding epistasis while keeping type-I errors to a minimum.

- **A parallel CPU application for exhaustive any-order epistasis detection that exploits all levels of parallelism of a homogeneous *x86_64* cluster.** This implementation combines a novel SIMD association-testing algorithm with a distributed execution to exploit the bit-level, data-level and task-level parallelism of a CPU cluster. The program includes explicit vector implementations of the MI association test for the *AVX2* and *AVX512BW* vector extensions of the *x86_64* architecture, as well as an autovectorization-friendly standard C++ implementation for different architectures. On top of that, the method implements a distributed any-order epistasis detection algorithm that exploits the CPU hardware resources of a cluster through MPI and multithreading to speed up the execution. The proposed program is, to our knowledge, faster than any other state-of-the-art method and supports a larger scope of application than the alternatives.

- **A new method for the calculation of high-order penetrance tables for bivariate epistasis models.** Previous penetrance table calculation methods tried to solve an extremely complex equation system made of the prevalence and heritability expressions, when the conditional probabilities of the genotypes based on the phenotype outcome were replaced in the equations by the bivariate model expressions. This approach produces incompatible equation systems, as certain values of prevalence are never reached within the system when the heritability is fixed to a particular level, and vice versa. In our approach, only one of the two parameters is fixed, while the other is maximized, dramatically reducing the risk of formulating an incompatible system. The result is a method capable of handling high order penetrance tables that resemble characteristics from real world

diseases more closely.

These contributions have been materialized in three different software programs: *MPI3SNP*, *Fiuncho* and *Toxo*. Both *MPI3SNP* and *Fiuncho* are distributed as source code and compiled through *CMake*, picking up the libraries for the system and using the appropriate implementation for the target architecture with little to no user intervention. The programs are run from command-line interface through MPI and the output is provided as a text file, similar to other MPI applications. *Toxo*, on the other hand, is a library written in MATLAB which provides several functions for reading an epistasis model from an input file, calculating the penetrance table from a set of input parameters and the model, and writing the calculated table to an output file in multiple formats. It is intended for audiences with basic programming skills, and it is a bit more complex to use than the previous two programs.

To further improve the ability of detecting epistasis interactions, there are several lines of work that can be continued in the future. Both *MPI3SNP* and *Fiuncho* only report the top-ranking combinations and the MI associated with them. Although this metric is appropriate to compare the degree of association between combinations, it is very difficult to judge if the association effect is significant only analyzing the MI value. Including a p-value in the final list of combinations could be a solution to this problem. Furthermore, the MI statistic does not distinguish between additive and non-additive effects of the epistasis interaction. Future work should also improve the association test to differentiate between additive and non-additive effects, similar to how Bayat et al. distinguish between the association power and the interaction effect in [92].

A different line of future work is related to the improvement of the detection power of non-exhaustive approaches. Current ones are not capable of locating epistasis interactions in the absence of marginal effects. Despite the advances in the exhaustive method presented in this thesis, this approach is unfeasible for genome-wide high-order epistasis analysis. Therefore, it is necessary to develop new non-exhaustive approaches focused on detecting high-order epistasis in datasets without marginal effects, while considering HPC techniques to accelerate their execution when needed.

As for *Toxo* (and *PyToxo*), future work includes expanding the scope of application even further. The current approach establishes the prevalence or heritability to a particular value while the other parameter is systematically maximized. However, once

the maximum prevalence or heritability is obtained, it is guaranteed that values under that maximum are attainable for the current equation system. Further improvements can be made so that after the maximum is obtained and the limits of the epistasis model are know, both prevalence and heritabilities can be simultaneously set (one using the original value and the other complying with the maximum value).

The results of the research carried out throughout this doctoral thesis have been published in the following international journals:

- C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Fiuncho: a program for any-order epistasis detection in CPU clusters». In: *The Journal of Supercomputing* (2022).
- B. González-Seoane, C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «PyToxo: a Python tool for calculating penetrance tables of high-order epistasis models». In: *BMC Bioinformatics* 23.117 (2022)
- C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «A SIMD algorithm for the detection of epistatic interactions of any order». In: *Future Generation Computer Systems* 132 (2022), pp. 108–123
- C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Evaluation of existing methods for high-order epistasis detection». In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.2 (2022), pp. 912–926
- C. Ponte-Fernández, J. González-Domínguez, A. Carvajal-Rodríguez, and M. J. Martín. «Toxo: a library for calculating penetrance tables of high-order epistasis models». In: *BMC Bioinformatics* 21.138 (2020)
- C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Fast search of third-order epistatic interactions on CPU and GPU clusters». In: *The International Journal of High Performance Computing Applications* 34.1 (2020), pp. 20–29

A short summary of the results of this thesis have been presented in the following poster session:

- C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Poster: Genome wide association studies in high performance computing systems». In: *17th International Summer School on Advanced Computer Architecture and Compilation for High-performance Embedded Systems*. Fiuggi (Italy), 2021

All presented programs and datasets are distributed as open source software, and they are available at GitHub:

- *MPI3SNP*:

  https://github.com/UDC-GAC/mpi3snp

- *Fiuncho*:

  https://github.com/UDC-GAC/fiuncho

- *Toxo*:

  https://github.com/UDC-GAC/toxo

- Datasets used in Chapter 3:

  https://github.com/UDC-GAC/epistasis-simulation-data

Appendix A

Program Configurations Used During the Survey Study

This appendix addresses how the relevant parameters for each of the methods were chosen. Formatting of the input data for each method will not be covered.

A.1. AntMiner

AntMiner is a method of the swarm intelligence family and as such, the same number of 100 iterations and 500 agents were used across methods. In *AntMiner*, however, there is no specific parameter to set the number of iterations to 100. The number of iterations is equal to the number of sub-colonies multiplied by the number of base iterations. At the same time, the number of sub-colonies is equal to the size of the iteration we are looking for. Therefore, the number of base iterations has to be set to 33 and 25 for third and fourth-order searches respectively. All other parameters were left with their by-default values.

GUI options

PARAMETER	THIRD ORDER	FOURTH ORDER
Ant Number	500	500
Base Iterations	33	25
Ant Sub-colonies	3	4
Initial Pheromones	100	100
Evaporation Rate	0.05	0.05
Pher Importance	1	1
Heur Importance	1	1
Significance level	0.05	0.05

A.2. ATHENA

ATHENA was executed using the grammatical evolution neural network algorithm following the sample configuration file `casecon.GENN` available with the tool, using a population size of 400 individuals in each of 10 demes during 400 generations, with crossover and mutation probabilities of 0.9 and 0.01, respectively.

Grammar files

The grammar files used for third and fourth-order searches are slightly modified versions of the original grammar file `genn_add.gram` included with the application. These grammar files have been verified by the maintainers of the application to ensure correct behaviour.

Third order

```
1 <p>       ::= <pn>(<pinput>)
2 <pn>      ::= PA
3 <pinput>  ::=  <W>(<winput>)<,><W>(<winput>)<,><W>(<winput>)<,>3
4 <winput>  ::= <cop><,><v>
5               | <cop><,><p>
6 <cop>     ::= (<cop><op><cop>)
7               | <Concat>(<num>)
```

```
<Concat> ::= Concat
<,>      ::= ,
<W>      ::= W
<op>     ::= +
             | -
             | *
             | /
<num>    ::=  <dig>1
             |  <dig>.<dig>3
             |  .<dig><dig>3
             |  <dig>.<dig><dig>4
             |  <dig><dig>.<dig><dig>5
<dig>    ::= 0
             | 1
             | 2
             | 3
             | 4
             | 5
             | 6
             | 7
             | 8
             | 9
<v>      ::= G1-4
```

Fourth order

```
<p>      ::= <pn>(<pinput>)
<pn>     ::= PA
<pinput> ::= <W>(<winput>)<,><W>(<winput>)<,><W>(<winput>)<,><W
    >(<winput>)<,>4
<winput> ::= <cop><,><v>
             | <cop><,><p>
<cop>    ::= (<cop><op><cop>)
             | <Concat>(<num>)
<Concat> ::= Concat
<,>      ::= ,
<W>      ::= W
<op>     ::= +
             | -
```

```
13              | *
14              | /
15  <num>    ::= <dig>1
16              | <dig>.<dig>3
17              | .<dig><dig>3
18              | <dig>.<dig><dig>4
19              | <dig><dig>.<dig><dig>5
20  <dig>    ::= 0
21              | 1
22              | 2
23              | 3
24              | 4
25              | 5
26              | 6
27              | 7
28              | 8
29              | 9
30  <v>      ::= G1-4
```

Configuration files

Third order

```
1   ALGORITHM GENN
2   MAXDEPTH 10
3   SENSIBLEINIT TRUE
4   POPSIZE 400
5   PROBCROSS 0.9
6   PROBMUT 0.01
7   GRAMMARFILE genn_add_3.gram
8   CALCTYPE RSQUARED
9   EFFECTIVEXO TRUE
10  GENSPERSTEP 400
11  INCLUDEALLSNPS TRUE
12  BACKPROPFREQ 100
13  BACKPROPSTART 0
14  END GENN
15
16  DATASET data.txt
```

Fourth order

```
1   ALGORITHM GENN
2   MAXDEPTH 10
3   SENSIBLEINIT TRUE
4   POPSIZE 400
5   PROBCROSS 0.9
6   PROBMUT 0.01
7   GRAMMARFILE genn_add_4.gram
8   CALCTYPE RSQUARED
9   EFFECTIVEXO TRUE
10  GENSPERSTEP 400
11  INCLUDEALLSNPS TRUE
12  BACKPROPFREQ 100
13  BACKPROPSTART 0
14  END GENN
15
16  DATASET data.txt
```

```
17 IDINCLUDED FALSE
18 MISSINGVALUE -9
19 CONTINMISS -9
20 DUMMYENCODE STEPHEN
21 RANDSEED 1
22 CV 5
23 NUMSTEPS 10
24 WRITECV FALSE
25 SUMMARYONLY TRUE
26 LOG NONE
27 INDOUTPUT TRUE
```

```
17 IDINCLUDED FALSE
18 MISSINGVALUE -9
19 CONTINMISS -9
20 DUMMYENCODE STEPHEN
21 RANDSEED 1
22 CV 5
23 NUMSTEPS 10
24 WRITECV FALSE
25 SUMMARYONLY TRUE
26 LOG NONE
27 INDOUTPUT TRUE
```

A.3. BADTrees

BADTrees was run setting the number of bootstrap samples to 0.01 and the number of nodes on each tree to 25.

Command line arguments

Invocation is the same for third and fourth order interactions:

```
1 ./snplash -engine adtree -phen data.covphen -geno data.geno \
2   -map data.map --bags 100 --nodes 25
```

A.4. BEAM3

BEAM3 only requires the prior probability common to all SNPs, calculated as the order of interaction divided by the total number of SNPs in the dataset. In our case, the prior probability for a third-order search is 0.006, and for a fourth-order search is 0.008.

Command line arguments

Third order

```
./BEAM3 data.txt -o out.txt\
  -prior 0.006
```

Fourth order

```
./BEAM3 data.txt -o out.txt\
    -prior 0.008
```

A.5. BHIT

With *BHIT* we used 30 000 Markov chain Monte Carlo iterations and 29 000 burn-in iterations, and specified the number of individuals, the number of SNPs and the average MAF of all SNPs. Since our SNP MAFs are uniformly sampled from the interval $[0.05, 0.5]$, the average MAF will be $(0.05 + 0.5)/2 = 0.275$.

Command line arguments

Invocation is the same, regardless of the order of the interactions:

```
./BHIT data.txt out.txt 30000 29000 2000 500 1 0.275 1
```

A.6. CINOEDV

CINOEDV is a method of the swarm intelligence family and as such, the same number of 100 iterations and 500 agents were used across methods. The interaction size was provided, and the accelerations of the particles, the evaluation measure and the alpha value were left unchanged from the values of its original paper.

R code snippet

Third order

```
Effect <- PSOSearch(pts=data$pts, class=data$class, MaxOrder=3,
  Population=500, Iteration=100, c1=2, c2=2, TopSNP=100,
```

```
  measure=2, alpha=0)
Effect <- NormalizationEffect(MaxOrder=3,
  SingleEffect=Effect$SingleEffect, TwoEffect=Effect$TwoEffect,
  ThreeEffect=Effect$ThreeEffect, FourEffect=Effect$FourEffect,
  FiveEffect=Effect$FiveEffect)
Effect <- NotationName(MaxOrder=3,
  SingleEffect=Effect$SingleEffect, TwoEffect=Effect$TwoEffect,
  ThreeEffect=Effect$ThreeEffect, FourEffect=Effect$FourEffect,
  FiveEffect=Effect$FiveEffect, SNPNames=data$names)
```

Fourth order

```
Effect <- PSOSearch(pts=data$pts, class=data$class, MaxOrder=4,
  Population=500, Iteration=100, c1=2, c2=2, TopSNP=100,
  measure=2, alpha=0)
Effect <- NormalizationEffect(MaxOrder=4,
  SingleEffect=Effect$SingleEffect, TwoEffect=Effect$TwoEffect,
  ThreeEffect=Effect$ThreeEffect,  FourEffect=Effect$FourEffect,
  FiveEffect=Effect$FiveEffect)
Effect <- NotationName(MaxOrder=4,
  SingleEffect=Effect$SingleEffect, TwoEffect=Effect$TwoEffect,
  ThreeEffect=Effect$ThreeEffect, FourEffect=Effect$FourEffect,
  FiveEffect=Effect$FiveEffect, SNPNames=data$names)
```

A.7. DCHE

In *DCHE* we indicated the maximum order of epistasis, the Bonferroni-corrected p-value threshold of 0.05 (resulting in 7.0×10^{-4} for second-order combinations, 5.0×10^{-9} for third-order combinations and 4.0×10^{-22} for fourth-order combinations) and to retain 10000, 4000 and 100 top-ranking combinations for the second, third and fourth orders, respectively.

Configuration files

Third order

```
1 [NO.SAMPLES] 2000
2 [NO.CASES] 1000
3 [NO.SNPS] 500
4 [ORDER] 3
5 [ALPHA0] 7.0E-4, 5.0E-9
6 [SIZELIST] 10000, 100
```

Fourth order

```
1 [NO.SAMPLES] 2000
2 [NO.CASES] 1000
3 [NO.SNPS] 500
4 [ORDER] 4
5 [ALPHA0] 7.0E-4, 5.0E-9, 4.0
      E-22
6 [SIZELIST] 10000, 4000, 100
```

A.8. EACO

EACO is a method of the swarm intelligence family and as such, the same number of 100 iterations and 500 agents are used across methods. Other than that, the interaction order and an evaporation rate of 0.3 were specified.

MATLAB code definitions

Interaction order can only be specified by changing a variable named `Dimension` inside of the *EACO* function (Line 30):

Third order

```
1 Dimension = 3;
```

Fourth order

```
1 Dimension = 4;
```

MATLAB code snippet

Since interaction order is established on an internal variable, the function invocation is the same regardless of the order:

```
1 [Result, Value, Time] = EACO('data.txt', 500, 100, 0.3);
```

A.9. EDCF

For *EDCF* we specified the maximum epistasis size, a number of 2500 retained top-ranking combinations, a permutation test p-value threshold of 0.05 and the Bonferroni-corrected p-value thresholds of 2.41×10^{-10} and 7.77×10^{-15} for third and fourth-order combinations, respectively.

Configuration files

Third order

```
[DISEASEN] 3
[MULFACTR] 25
[MAX_PVAL] 2.41E-10
[MAXGWPVL] -1
[ALPHA_RF] 0.05
```

Fourth order

```
[DISEASEN] 4
[MULFACTR] 25
[MAX_PVAL] 7.77E-15
[MAXGWPVL] -1
[ALPHA_RF] 0.05
```

Command line arguments

Third order

```
./edcflinux_64bit 100
```

Fourth order

```
./edcflinux_64bit 100
```

A.10. epiACO

epiACO is a method of the swarm intelligence family and as such, the same number of 100 iterations and 500 agents were used across methods. Additionally, the interaction order and an evaporation rate of 0.2 were specified.

MATLAB code definitions

Interaction order can only be specified by changing a variable named `Dimension` inside the *epiACO* function (Line 32):

Third order

```
1 Dimension = 3;
```

Fourth order

```
1 Dimension = 4;
```

MATLAB code snippet

Since interaction order is established on an internal variable, the function invocation is the same regardless of the order of the interaction:

```
1 [Result, Value, Time] = epiACO('data.txt', 500, 100, 0.2);
```

A.11. EpiMiner

EpiMiner was run using the exact calculation of the co-information index and letting the SVM decide the number of SNPs to retain from the filtering step. We refrained from using Monte Carlo sampling to approximate the index, since the authors do not suggest any number of samples to use and we found no correlation between this number and the runtime of the program.

GUI options

PARAMETER	THIRD ORDER	FOURTH ORDER
Max Dimension	3	4
Enumeration	Automatic	Automatic

A.12. FDHE-IW

For *FDHE-IW* we specified the maximum epistasis order present in the data, a *p*-value threshold of 0.05 and the average MAF of all SNPs of $(0.05+0.5)/2 = 0.275$.

MATLAB code snippet

Third order

```
data = dlmread('data.txt','\t', 1, 0);
[~,col]= size(data);
Dim = col - 1;
epi_dim = 3;
MAF = 0.275;
pvalue = 0.05 / nchoosek(Dim, epi_dim) / MAF;
col = col - 1;
su = zeros(1,col);
hx = zeros(1,col);
state = data(:,col + 1);
tm = 2;
K = epi_dim;
w = ones(1,col);
for i = 1:col
  Hxy(i) =jointEntropy(data(:,[i,col+1]))+hx(i);
  su(i) = SU2(data(:,i),state, Hxy(i));
end

su2 = su;
CandidateSize =4;
S = [];
F = 1:col;
w(1:col) = 1;
selSize = 0;
Candidate = [];
combResult = [];
pvalResult = [];
while selSize<CandidateSize
  if ~isempty(Candidate)
```

```
    CsSize = length(Candidate(:,1));
  else
    CsSize = 0;
  end
  Candidate = iwf(S,w,K,su2,tm,F,data,state,Candidate);
  w(Candidate(1:length(Candidate(:,1)),1)) = 0;
  selSize = selSize + length(Candidate(:,1)) - CsSize;
end

for r = 1:CandidateSize + 1
  S = Candidate(r,:);
  S = sort(S);
  [P_value] = Gtest_score(data(:,S),data(:,Dim+1));
  if P_value < pvalue
    [P_value2] = permutation(data(:,S),data(:,Dim+1),100,
                             pvalue);
    if P_value2 < pval
      combResult = [combResult; S];
      pvalResult = [pvalResult; P_value2];
    end
  end
end
```

Fourth order

```
data = dlmread('data.txt','\t', 1, 0);
[~,col]= size(data);
Dim = col - 1;
epi_dim = 4;
MAF = 0.275;
pvalue = 0.05 / nchoosek(Dim, epi_dim) / MAF;
col = col - 1;
su = zeros(1,col);
hx = zeros(1,col);
state = data(:,col + 1);
tm = 2;
K = epi_dim;
w = ones(1,col);
for i = 1:col
```

```
15    Hxy(i) =jointEntropy(data(:,[i,col+1]))+hx(i);
16    su(i) = SU2(data(:,i),state, Hxy(i));
17  end
18
19  su2 = su;
20  CandidateSize =4;
21  S = [];
22  F = 1:col;
23  w(1:col) = 1;
24  selSize = 0;
25  Candidate = [];
26  combResult = [];
27  pvalResult = [];
28  while selSize<CandidateSize
29    if ~isempty(Candidate)
30      CsSize = length(Candidate(:,1));
31    else
32      CsSize = 0;
33    end
34    Candidate = iwf(S,w,K,su2,tm,F,data,state,Candidate);
35    w(Candidate(1:length(Candidate(:,1)),1)) = 0;
36    selSize = selSize + length(Candidate(:,1)) - CsSize;
37  end
38
39  for r = 1:CandidateSize + 1
40    S = Candidate(r,:);
41    S = sort(S);
42    [P_value] = Gtest_score(data(:,S),data(:,Dim+1));
43    if P_value < pvalue
44      [P_value2] = permutation(data(:,S),data(:,Dim+1),100,
45                               pvalue);
46      if P_value2 < pval
47        combResult = [combResult; S];
48        pvalResult = [pvalResult; P_value2];
49      end
50    end
51  end
```

A.13. GALE

GALE was run using the same parameters as in the original paper: 10 cross-validation partitions, 2500 iterations, a 25 by 25 grid size, a uniform resource allocation, allowing rule pruning and a probability of wild incorporation of 0.75.

Command line arguments

Command invocation is equal for both interaction orders:

```
python2 ./GALE_Main.py gh data.txt data.txt progress output \
   1 10 1 2500 25 0 1 0.75
```

A.14. HiSeeker

During the *HiSeeker* evaluation, a *p*-value threshold of 0.05 and a relaxation scale factor of 10000 were used, and the order of the epistasis interaction was provided. Both ACO and exhaustive alternatives of *HiSeeker* were run, and for the ACO version we modified the number of ants to 500 and the iterations to 100 in order to match the rest of ACO methods; all other parameters were left with their default values.

Configuration files

Third order – ACO

```
input_file: data.txt
threshold: 0.05
the_scale_factor: 10000
rou: 0.05
phe: 100.00
alpha: 1.00
iAntCount: 500
iterCount: 100
```

Fourth order – ACO

```
input_file: data.txt
threshold: 0.05
the_scale_factor: 10000
rou: 0.05
phe: 100.00
alpha: 1.00
iAntCount: 500
iterCount: 100
```

```
kLociSet:   2
kEpiModel: 3
kTopModel: 1000
topK: 100
typeOfSearch: 1
numberOfthread: 1
```

```
kLociSet:   2
kEpiModel: 4
kTopModel: 1000
topK: 100
typeOfSearch: 1
numberOfthread: 1
```

Third order – Exhaustive

```
input_file: data.txt
threshold: 0.05
the_scale_factor: 10000
rou: 0.05
phe: 100.00
alpha: 1.00
iAntCount: 500
iterCount: 100
kLociSet:   2
kEpiModel: 3
kTopModel: 1000
topK: 100
typeOfSearch: 0
numberOfthread: 1
```

Fourth order – Exhaustive

```
input_file: data.txt
threshold: 0.05
the_scale_factor: 10000
rou: 0.05
phe: 100.00
alpha: 1.00
iAntCount: 500
iterCount: 100
kLociSet:   2
kEpiModel: 4
kTopModel: 1000
topK: 100
typeOfSearch: 0
numberOfthread: 1
```

A.15. IACO

IACO is a method of the swarm intelligence family and as such, the same number of iterations and agents were used across methods. For *IACO* we also indicated the interaction order.

MATLAB code definitions

Interaction order can only be specified by changing a variable named `Dimension` inside the *IACO* function (Line 26):

Third order

```
Dimension = 3;
```

Fourth order

```
Dimension = 4;
```

MATLAB code snippet

```
[Result, Value, Time] = IACO('data.txt', 500, 100);
```

A.16. LAMPLINK

LAMPLINK only requires to specify the variation range of the MAF $[0.05, 0.5]$ common to all SNPs. Additionally, we also use a recessive model following the authors' recommendations, since using a dominant model leads to large runtimes when paired with high MAF values.

Command line arguments

Command invocation is equal for both interaction orders:

```
./lamplink --file 'data.txt' --allow-no-sex --lamp \
  --model-rec --sglev 0.05 --upper 0.5 --out output.txt
```

A.17. LRMW

LRMW barely needs any parameterization, only requiring the maximum order of combinations to explore.

Command line arguments

Third order

```
./gwggi --file data.txt \
  --lmw --hz --hiorder 3 \
  --out output.txt
```

Fourth order

```
./gwggi --file data.txt \
  --lmw --hz --hiorder 4 \
  --out output.txt
```

A.18. MACOED

MACOED is a method of the swarm intelligence family and as such, the same number of iterations and agents were used across methods. *MACOED* additionally requires the epistasis order and the Bonferroni corrected value of the p-value 0.05, resulting in the p-values of 2.4145×10^{-9} and 1.9432×10^{-11} for third and fourth-order combinations, respectively.

MATLAB code snippet

Third order

```
filter_snps = acomop_seek(data.Variables, 3, 500, 100);
adj_pvalue = 0.05/nchoosek(size(data, 2) - 1, 3);
epistatic = Chisuqare_test(data.Variables, filter_snps, 3,
                           adj_pvalue);
```

Fourth order

```
filter_snps = acomop_seek(data.Variables, 4, 500, 100);
adj_pvalue = 0.05/nchoosek(size(data, 2) - 1, 4);
epistatic = Chisuqare_test(data.Variables, filter_snps, 4,
                           adj_pvalue);
```

A.19. MDR

The combination search size in *MDR* was restricted to those of the same size as the epistatic combination included in the data. 10 cross-validation partitions were used according to its original evaluation.

Command line arguments

Third order

```
1 java -jar ./mdr_3.0.2.jar -min=3 -max=3 -cv=10 -table_data=true\
2   -nolandscape -top_models_landscape_size=100 data.txt
```

Fourth order

```
1 java -jar ./mdr_3.0.2.jar -min=4 -max=4 -cv=10 -table_data=true\
2   -nolandscape -top_models_landscape_size=100 data.txt
```

A.20. MECPM

MECPM was executed using the greedy approach, with the default pool size of 25 and indicating the order of the target epistatic combination. The rest of the parameters were left with their by-default values.

C code definitions

MECPM parameters can only be changed inside the C header file `mecpm.h` (Lines 4–15):

Third order

Fourth order

```
#define dim 500
#define max_num_pts 2000
#define Ncon 25
#define max_degree 3
#define max_constraints 10
#define num_iters 10
#define num_codes 2
#define num_classes 2
#define like_thresh 1e-3
#define eps 1e-10
```

```
#define dim 500
#define max_num_pts 2000
#define Ncon 25
#define max_degree 4
#define max_constraints 10
#define num_iters 10
#define num_codes 2
#define num_classes 2
#define like_thresh 1e-3
#define eps 1e-10
```

A.21. Mendel

Mendel method description includes three different stages, however authors only use two during its evaluation: the first filtering stage and the last interaction selection stage. To replicate that, we took one of the example configuration files provided with the tool (`Control24c.in`) and disabled the `Penalized_Regression` option, set the number of predictors in each of the stages as 10 and 20 (since they are the best choice according to the authors) and disabled the group weighting because it is not used (nor mentioned) in the original evaluation.

Configuration files

Third order

```
Definition_file = Def.in
Pedigree_file = Ped.in
SNP_definition_file = SNP_def.in
SNP_data_file = SNP_data.in
Case2_Control1 = True
Summary_file = Summary.out
Analysis_option = GWAS
Model = 2
Min_Success_Rate_Per_SNP = 0
Min_Success_Rate_Per_Individual = 0
```

```
11 Quantitative_Trait = Status
12 Marginal_Analysis = True
13 Penalized_Regression = False
14 Penalized_Interaction = True
15 Interaction_Levels = 3
16 Desired_Predictors = 10 :: Marginal
17 Desired_Predictors = 20 :: Interactions
18 Uniform_Weights = True
```

Fourth order

```
1  Definition_file = Def.in
2  Pedigree_file = Ped.in
3  SNP_definition_file = SNP_def.in
4  SNP_data_file = SNP_data.in
5  Case2_Control1 = True
6  Summary_file = Summary.out
7  Analysis_option = GWAS
8  Model = 2
9  Min_Success_Rate_Per_SNP = 0
10 Min_Success_Rate_Per_Individual = 0
11 Quantitative_Trait = Status
12 Marginal_Analysis = True
13 Penalized_Regression = False
14 Penalized_Interaction = True
15 Interaction_Levels = 4
16 Desired_Predictors = 10 :: Marginal
17 Desired_Predictors = 20 :: Interactions
18 Uniform_Weights = True
```

A.22. MPI3SNP

In our modified version of *MPI3SNP*, the combination size can be specified as a command-line argument to restrict the search space to combinations of a given size. Additionally, we indicated to use a single thread during the execution and to output 10 combinations, following the advice of [40].

Command line arguments

Third order

```
mpiexec -n 1 ./mpi3snp -t 1\
  -s 3 -n 10 data.tped \
  data.tfam output.txt
```

Fourth order

```
mpiexec -n 1 ./mpi3snp -t 1\
  -s 4 -n 10 data.tped \
  data.tfam output.txt
```

A.23. NHSA-DHSC

In *NHSA-DHSC* the recommended number of agents and iterations is far off the values of the rest of the swarm intelligence methods, and therefore those were used. We specified the order of the interaction, a harmony memory size of 100, a maximum number of iterations of 60 000, a maximum number of local search iterations of 2000, a candidate memory size of 10 per metric and Bonferroni-corrected p-value thresholds of 2.4145×10^{-9} and 1.9432×10^{-11} for third and fourth order, respectively.

MATLAB code snippet

Third order

```
epi_dim = 3;
Dim = size(data, 2) - 1;
pvalue = 0.05 / nchoosek(Dim, epi_dim);
CX = Dim - epi_dim + 1 : Dim;
[Candidate, canSize, ~, ~, ~] = NHSA3(
  data, epi_dim, 100, 60000, epi_dim * 500, 10, CX);
G_set = [];
for i = 1:canSize
  c = Candidate(i, 1:epi_dim);
  P_value = Gtest_score(c, data(:, Dim+1));
  if P_value < pvalue
    [p, ~] = permutation(data(:, c), data(:,Dim+1), 100,
                         pvalue, 0);
    G_set = [G_set; c p];
```

```
15    end
16  end
```

Fourth order

```
1  epi_dim = 4;
2  Dim = size(data, 2) - 1;
3  pvalue = 0.05 / nchoosek(Dim, epi_dim);
4  CX = Dim - epi_dim + 1 : Dim;
5  [Candidate, canSize, ~, ~, ~] = NHSA3(
6    data, epi_dim, 100, 60000, epi_dim * 500, 10, CX);
7  G_set = [];
8  for i = 1:canSize
9    c = Candidate(i, 1:epi_dim);
10   P_value = Gtest_score(c, data(:, Dim+1));
11   if P_value < pvalue
12     [p, ~] = permutation(data(:, c), data(:,Dim+1), 100,
13                          pvalue, 0);
14     G_set = [G_set; c p];
15   end
16 end
```

A.24. SingleMI

In *SingleMI* we specified the maximum combination size, and set the number of candidate SNPs to be extracted from the clusters equal to 50.

Command line arguments

Third order

```
1  ./singlemi -p data.tped \
2    -f data.tfam -c 50 -k 3
```

Fourth order

```
1  ./singlemi -p data.tped \
2    -f data.tfam -c 50 -k 4
```

A.25. SNPHarvester

For *SNPHarvester* we followed the parameterization used during its original evaluation, only modifying the minimum and maximum combination size to indicate the interaction order and increase the *p*-value to 0.05 to match the rest of the methods.

Java code definitions

SNPHarvester parameters can only be changed inside the Java source code file `SNPHarvester.java` (Lines 785-795):

Third order

```
String path = "data.txt";
String measure =
  "chi_square";
String resultType =
  "thresholdBased";
int numSuccessiveRun = 3;
double pValue = 0.05;
int maxk = 3;
int mink = 3;
int topK = 5;
```

Fourth order

```
String path = "data.txt";
String measure =
  "chi_square";
String resultType =
  "thresholdBased";
int numSuccessiveRun = 3;
double pValue = 0.05;
int maxk = 4;
int mink = 4;
int topK = 5;
```

A.26. SNPRuler

SNPRuler was used with a search tree size of 50 000, a maximum depth equal to the epistatic interaction order and a utility threshold for rule updating of 0.001.

Command line arguments

Third order

Fourth order

```
java -jar ./rule.jar 50000 \
   3 0.001 data.txt
```

```
java -jar ./rule.jar 50000 \
   4 0.001 data.txt
```

A.27. StepPLR

With *StepPLR* we replicated the evaluation parameters used in [108], leaving the default penalization coefficient λ of 1×10^{-4}, and using the default BIC cost function in the SNP selection process with both forward and backward selection steps.

R code snippet

The code is equal for both interaction orders:

```
fit = step.plr(x, y, trace = FALSE)
```

Apéndice B

Extended Summary in Spanish

En este apéndice se incluye un resumen más extenso de la tesis. Está estructurado en un total de siete secciones. En primer lugar se introducen algunos conceptos previos y se explica el test de asociación empleado a lo largo del trabajo. A continuación presentamos una herramienta para la detección de epistasia de tercer orden en clústeres de CPUs y GPUs. A esto le sigue una revisión y evaluación de las herramientas existentes en el estado del arte. Una vez hecho esto, presentamos una implementación vectorial del test de asociación, y a su vez incluimos esta implementación en una herramienta paralela capaz de detectar epistasia de cualquier orden en clústeres de CPUs. Por último, discutimos las conclusiones de esta tesis, el trabajo futuro y los resultados de la investigación realizada.

B.1. Introducción

Esta tesis se centra en la mejora de las herramientas dedicadas a la detección de la epistasia en estudios de asociación del genoma completo (GWAS, por sus siglas en inglés). Los GWAS son estudios que buscan establecer una relación entre marcadores genómicos y un fenotipo de interés. Estos estudios son especialmente importantes porque nos permiten obtener información de una enfermedad para entender su funcionamiento y poder prevenirla, diagnosticarla y tratarla. Pese a la gran cantidad de variantes genéticas identificadas como asociadas a determinados fenotipos [14], es-

tos estudios no dieron los resultados esperados. Una de las hipótesis para esta falta de resultados es la epistasia, la interacción entre genes que enmascaran dichas asociaciones. En este trabajo nos centramos en la epistasia desde una perspectiva estadística, mediante la cual se define la epistasia como la desviación de la expresión de un fenotipo como la suma de efectos individuales de los genes que intervienen.

La detección de epistasia computacionalmente supone encontrar la combinación, o combinaciones, de variantes correctas de entre $\binom{n}{k}$ posibilidades, siendo n el número de variantes en cuestión y k el tamaño de las combinaciones, conocido también como orden de la interacción. Motivados por este crecimiento exponencial del problema, los autores de este campo se decantaron por aproximaciones que se pueden dividir en dos grupos:

- Métodos exhaustivos. Es una aproximación de fuerza bruta, donde se recorren todas las combinaciones de k variantes posibles y se identifican aquellas que presentan asociación con el rasgo de interés. Estos métodos tienen una complejidad temporal de $O(n^k \cdot A)$, siendo A la complejidad del test de asociación que se repite para cada combinación. Esta complejidad asume que el número de combinaciones, $\binom{n}{k}$, es equivalente a n^k, ya que $k < n - k$. Las implementaciones disponibles en la bibliografía se centran en mejorar el rendimiento y se diferencian principalmente en la elección de los aceleradores o la infraestructura de computación de altas prestaciones (HPC, por sus siglas en inglés) utilizada.
- Métodos no exhaustivos. Siguen una heurística para guiar la búsqueda, evitando recorrer todas las combinaciones. El enfoque de este segundo grupo de métodos es más algorítmico, con aproximaciones con complejidades muy diferentes. Dentro de este grupo se puede hacer una distinción adicional: por una parte tenemos métodos que combinan conocimiento biológico para filtrar el total de combinaciones a analizar, y por otro lado tenemos aquellas herramientas centradas en el análisis estricto de los datos y que se alimentan de una multitud de campos como la teoría de la información, el aprendizaje automático, análisis de regresión, etc.

Independientemente de la aproximación, todos los métodos necesitan una forma de cuantificar el grado de asociación entre una combinación de genotipos y el fenotipo

de estudio. En esta tesis hemos usado el mismo test de asociación a lo largo de todo el trabajo. Es un test basado en la Información Mutua (IM), y que se compone de tres partes:

1. La representación de la información genética en tablas de genotipos cuya función es, además de la propia representación en formato binario, la agrupación de los individuos en función de sus características genotípicas y fenotípicas.
2. La caracterización de las frecuencias genotípicas atendiendo al fenotipo, a través de una tabla de contingencia.
3. El cálculo de la IM a partir de la tabla de contingencia anterior.

Considerando lo anterior, la primera tarea fue la creación de una herramienta exhaustiva de detección de epistasia basada en la IM, a partir de la cual evaluar el estado del arte y continuar mejorando el rendimiento de la misma, tanto desde una perspectiva de eficiencia computacional como de capacidad de detección de epistasia.

B.2. Búsqueda de Epistasia de Tercer Orden en clústeres de CPUs y GPUs

La detección de epistasia, al igual que muchos otros campos de la bioinformática, es un problema para el que muchos autores han recurrido al uso de HPC con el objetivo de acelerar el cómputo y poder abarcar problemas de dimensiones superiores. En nuestro caso, en la búsqueda de epistasia, nuestra primera aproximación fue la implementación de una búsqueda exhaustiva de interacciones de tres variantes. La herramienta, bautizada *MPI3SNP*, aprovecha las CPUs o GPUs disponibles en un clúster a través de la librería MPI, en combinación con multithreading y CUDA.

De cara a paralelizar la tarea, establecimos como unidad de reparto las diferentes combinaciones de dos variantes. Aunque pueda parecer anti-intuitivo, el reparto de pares garantiza que no haya solape entre las operaciones asignadas a cada unidad de

procesamiento (núcleos para la implementación CPU, o tarjetas gráficas en la implementación GPU). Estas unidades analizan todas las combinaciones de tres variantes que comienzan por los pares asignados, obteniendo las tablas de contingencia y calculando la IM asociada. Los pares son repartidos mediante una distribución cíclica a todas las unidades de cómputo (núcleos de CPUs o tarjetas GPUs), y el programa devuelve una lista de las combinaciones cuyo valor de IM es el más alto de entre todas.

La evaluación experimental de *MPI3SNP* demuestra tanto el balanceo de carga óptimo obtenido con la estrategia de reparto como la eficiencia paralela en el uso de recursos. Por una parte, hemos medido la diferencia de combinaciones de tres variantes asignadas a cada unidad de cómputo con respecto a la asignación ideal, y hemos observado que en el peor de los casos esta diferencia solo llega a ser del 3 %. Por otra parte, hemos evaluado la eficiencia paralela conforme aumentamos el número de recursos usados, tanto para la implementación CPU como la de GPU. Los resultados muestran unas eficiencias que se aproximan a la ideal.

B.3. Evaluación de Métodos de Detección de Epistasia de Orden Alto

Una vez diseñada una herramienta exhaustiva que nos sirva como punto de partida, procedimos a evaluar el estado del arte de la detección de epistasia de forma experimental, con un énfasis en la detección de interacciones de orden superior. Esta evaluación incluye veintiséis herramientas adicionales, y las compara en términos de tiempo de ejecución, poder de detección de epistasia y presencia de falsos positivos en la salida.

La primera mitad del estudio compara las herramientas en cuanto a aproximación, discutiendo las semejanzas y diferencias entre ellas. El estudio comienza agrupando los diferentes métodos en seis grupos, atendiendo a cómo efectúan la búsqueda:

- Métodos exhaustivos. En este grupo encontramos a *MDR* [70] y *MPI3SNP* [2].
- Métodos de filtrado. Estos métodos se caracterizan por eliminar polimorfismos de un solo nucleótido (SNP, de sus siglas en inglés), o combinaciones que co-

mienzan por algunas combinaciones de dos SNPs particulares, antes de comenzar la búsqueda. Para ello emplean un estadístico que determina la relevancia de ese SNP o par de SNPs, evitando explorar combinaciones de orden superior que los incluyan. *DCHE* [37], *EDCF* [62], *EpiMiner* [29], *HiSeeker* [66], *LAMPLINK* [68], *MECPM* [71], *Mendel* [72] y *SingleMI* [30] siguen este procedimiento.

- Métodos de búsqueda en profundidad. Exploran el espacio de búsqueda incorporando SNPs al grupo seleccionado a la vez que se maximiza alguna métrica. Estos métodos terminan cuando obtienen un número fijo de combinaciones, o bien ya no logran encontrar ninguna otra que consideren asociada. A este grupo pertenecen *BADTrees* [57], *FDHE-IW* [64], *LRWM* [69], *SNPRuler* [75] y *StepPLR* [33].

- Métodos de inteligencia de enjambre. Usan un algoritmo de inteligencia de enjambre para guiar la búsqueda de epistasia. Aquí nos encontramos a *AntMiner* [56], *CINOEDV* [60], *EACO* [61], *epiACO* [63], *IACO* [67], *MACOED* [32] y *NHSA-DHSC* [73].

- Algoritmos genéticos. Emplean algoritmos genéticos para converger hacia una combinación de variantes que explique las diferencias fenotípicas entre los diferentes individuos. *ATHENA* [31] y *GALE* [65] pertenecen a este grupo.

- Métodos de búsqueda aleatoria. Exploran el espacio de soluciones mediante un muestreo estocástico, evaluando la asociación de cada combinación explorada y devolviendo aquellas que mejor fitness obtienen. *BEAM3* [58], *BHIT* [59] y *SNPHarvester* [74] siguen este procedimiento.

La segunda mitad del estudio se centra en la evaluación experimental de las herramientas. Para ello empleamos conjuntos de datos sintéticos, pues la simulación ofrece un escenario donde poder controlar todos los parámetros poblacionales a nuestro antojo (como la frecuencia del alelo menos común, abreviada como MAF de sus siglas en inglés, de los SNPs; o la penetrancia o heredabilidad del fenotipo) y conocer de antemano la solución al problema de la búsqueda de epistasia. Para este propósito desarrollamos 55 modelos de epistasia empleando unos parámetros poblacionales inspirados

en los de enfermedades humanas, a partir de los cuales simulamos 5500 conjuntos de datos que suman un total de 2,75 millones de SNPs y 11 millones de individuos.

En primer lugar, la evaluación del tiempo de ejecución confirma la hipótesis de partida: el aumento del tiempo de ejecución con el orden es más acusado en las herramientas exhaustivas que en las no exhaustivas. En segundo lugar, en cuanto a poder de detección de epistasia, el estudio concluye que la única estrategia exitosa ante la ausencia de efectos marginales en las interacciones epistáticas es la exhaustiva. Los métodos no exhaustivos son incapaces de localizar las interacciones cuando las variantes que la componen no muestran cierta asociación por si solas con el fenotipo de estudio. Por último, en cuanto a falsos positivos se refiere, todas las herramientas obtienen de forma generalizada malos resultados. De las veintisiete herramientas, tan solo *BEAM3* es capaz de minimizar la presencia de falsos positivos sin perder la capacidad de detección de epistasia.

B.4. Implementación Vectorial del Test de Asociación

Una conclusión que se puede sacar de la evaluación anterior particular al método de detección de epistasia basado en la IM es que obtiene un muy buen poder de detección independientemente del modelo u orden de las interacciones a localizar. Es por ello que el siguiente paso de esta tesis fue optimizar su implementación en arquitecturas CPU. Para ello proponemos una estrategia de vectorización y unas implementaciones concretas empleando *AVX Intrinsics*, tanto para un ancho de 256 bits como de 512.

Anteriormente habíamos descompuesto este test de asociación en tres partes: la representación de la información en tablas de genotipos, el consiguiente cálculo de las tablas de contingencia a partir de las anteriores, y el cálculo de la IM a partir de las últimas. Todas estas operaciones emplean bucles que iteran sobre las diferentes celdas de las tablas, bien para construir otras o para obtener finalmente el valor de IM. Es por ello que el test de asociación es un muy buen candidato a la vectorización, ya que estas operaciones repetitivas sobre celdas diferentes de las tablas siguen el paradigma SIMD (del inglés *Single Instruction Multiple Data*). Casi todas las operaciones involucradas disponen de una instrucción AVX dedicada (lógica booleana y operaciones aritméticas

que incluyen sumas y productos), o bien existe una librería estándar que la implementa (la función logaritmo). La única excepción es el caso de la operación *popcount*, que cuenta el número de 1's en una palabra binaria, y para la cual no existe instrucción AVX hasta la microarquitectura Intel Ice Lake. Este problema fue resuelto mediante el uso de una implementación software de la operación vectorial.

Además de la propia vectorización del test de asociación, también proponemos un algoritmo para la detección de epistasia que tiene en cuenta los cambios de frecuencia de los procesadores *x86_64* asociados al uso de instrucciones vectoriales. Dichos procesadores se caracterizan por bajar la frecuencia al operar con aritmética vectorial en punto flotante con respecto a la aritmética vectorial entera o booleana. Es por eso que dividimos la carga de trabajo en bloques de operaciones, separando la aritmética de punto flotante del resto y así no ralentizando la ejecución. Aunque no es el objetivo principal de este trabajo, este algoritmo ya implementa la búsqueda de epistasia de cualquier orden, pues la implementación del test de asociación lo permite simplemente variando cuántas veces combinamos las tablas de genotipos entre sí antes de calcular la tabla de contingencia.

La evaluación de este algoritmo vectorial compara varias implementaciones vectoriales explícitas, empleando varios anchos vectoriales del test de asociación, con los códigos vectorizados de forma automática por los compiladores GCC e ICC. Esta evaluación compara tanto las operaciones individuales que componen el test de asociación, como el algoritmo al completo. Los resultados revelan que la implementación explícita es superior a las que ofrecen los compiladores. En cuanto al ancho vectorial, el de 512 bits generalmente ofrece mejores resultados para todos los órdenes de interacción y número de individuos probados.

B.5. Detección de Epistasia de Cualquier Orden en clústeres de CPUs

Una vez implementado el test de asociación aprovechando todos los recursos que un núcleo de CPU nos puede ofrecer, es necesario revisitar el problema de la búsqueda de epistasia en una arquitectura clúster, pero esta vez considerando interacciones de cualquier orden. Aunque la búsqueda exhaustiva pueda parecer irrealizable para in-

teracciones de más de tres variantes dada la complejidad computacional del método, existen estudios en el estado del arte que se limitan a analizar unos pocos cientos de marcadores genéticos y que se pueden beneficiar de dicha aproximación.

El punto de partida es el mismo que con *MPI3SNP*: distribuir una lista de combinaciones, esta vez de tamaño variable k, entre todas las unidades de cómputo, que esta vez son los núcleos del procesador, evitando la repetición de cálculos en la medida de lo posible. La estrategia en esta ocasión consiste en repartir dichas combinaciones empleando los prefijos compuestos por $k-1$ variantes, de forma que cada unidad de procesamiento calcule la IM de todas las combinaciones que comiencen por dicho prefijo. Esta estrategia introduce un pequeño sobrecoste durante el cálculo de las tablas de genotipos para la búsqueda de interacciones de tamaño 4 o superior. El algoritmo distribuido tiene en cuenta el problema de frecuencias del procesador *x86_64* descrito en la sección anterior, y sigue la misma estrategia de separar la aritmética vectorial de punto flotante en un bloque de operaciones diferente de la aritmética booleana y de enteros, evitando la ralentización del segundo.

Esta nueva herramienta, llamada *Fiuncho*, fue evaluada tanto por separado como con otros métodos de búsqueda exhaustiva disponibles en el estado del arte. En primer lugar medimos la eficiencia de la estrategia de reparto, obteniendo la diferencia entre el número de combinaciones asignadas a cada núcleo de cómputo y la asignación ideal, empleando hasta 522 núcleos. Los resultados indican que en el peor de los casos esta diferencia solo llega a ser el 3 % del total de combinaciones asignado al núcleo. A continuación medimos el sobrecoste introducido en la implementación paralela con respecto a la secuencial por la repetición de cálculos para órdenes superiores a tres, y los resultados indican que es despreciable. El siguiente paso fue medir la eficiencia en el uso de recursos durante una ejecución paralela, llegando a emplear hasta 504 núcleos, y obteniendo una eficiencia paralela cercana al ideal. Por último, comparamos *Fiuncho* con su antecesor *MPI3SNP* para la búsqueda de interacciones de tres variantes, y con *BitEpi* y *MDR* para interacciones de entre dos y cuatro variantes. Los resultados muestran que *Fiuncho* es la herramienta más rápida, siendo en promedio siete veces más rápida que *MPI3SNP*, tres veces más rápida que *BitEpi* y 358 veces más rápida que *MDR*. Las diferencias con *BitEpi* y *MDR* podrían ser todavía mayores si decidiésemos emplear más recursos hardware, pues estas herramientas están limitadas a ejecuciones internodo.

B.6. Cálculo de Tablas de Penetrancia para Modelos de Orden Alto

El cálculo de las tablas de penetrancia empleadas para la simulación de una interacción epistática es, sin lugar a duda, el tema que más se aleja de los objetivos inicialmente planteados para esta tesis. Las tablas de penetrancia son aquellas tablas que describen la frecuencia poblacional de un rasgo fenotípico para cada valor genotípico que los individuos presentan. Estas tablas, junto a unos valores de MAF para las variantes que intervienen, permiten definir la interacción epistática al completo. Es por ello que los simuladores de epistasia las usan con frecuencia para definir la interacción.

No obstante, los simuladores del estado del arte presentan limitaciones a la hora de obtener dichas tablas para interacciones de orden alto y siguiendo un modelo de interacción particular. Es por ello que desarrollamos *Toxo*, una librería escrita en MATLAB que permite el cálculo de dichas tablas de forma que los simuladores existentes las puedan usar y, por consiguiente, simular dichas interacciones epistáticas. Mediante *Toxo* podemos obtener tablas de penetrancia para interacciones de orden alto usando la mayoría de los modelos de interacción de la literatura existente. Gracias a esta herramienta hemos sido capaces de generar interacciones de hasta ocho variantes, un número muy superior a lo visto hasta la fecha, y para valores de prevalencia y heredabilidad mucho más cercanos a los valores que presentan enfermedades reales en poblaciones humanas.

B.7. Conclusiones y Trabajo Futuro

En la última década, los estudios GWAS han conseguido identificar cientos de variantes genéticas asociadas a enfermedades u otros rasgos humanos. Pese al gran número, estas variantes solo explican un porcentaje muy pequeño de la heredabilidad observada en estos fenotipos. Una hipótesis que explica este suceso es la presencia de epistasia de alto orden durante la expresión de dichos rasgos. Es por este motivo que la detección de la epistasia sigue siendo un campo activo de investigación. En este contexto, la tesis aquí presentada ha hecho las siguientes contribuciones:

- **Una herramienta paralela para la búsqueda exhaustiva de epistasia de tercer orden en clústeres de CPUs o GPUs.** La implementación combina MPI con multithreading y CUDA para explotar las diferentes CPUs o GPUs disponibles en los nodos interconectados del clúster. La distribución paralela estática implementada no requiere comunicaciones entre las diferentes unidades de cómputo, salvo durante la inicialización y terminación del programa, lo que la convierte en ideal para analizar grandes conjuntos de datos.

- **Un estudio empírico de los métodos de detección de epistasia de alto orden existentes.** El estudio incluye veintisiete métodos, agrupados en seis categorías atendiendo a sus similitudes y diferencias en la forma de buscar las interacciones, y los evalúa de forma empírica en términos de tiempo de ejecución, poder de detección y presencia de falsos positivos empleando un conjunto de datos común a todos los métodos. Sin duda, las conclusiones más relevantes de este estudio son las diferencias en capacidad de detección de epistasia ante la presencia y ausencia de efectos marginales en la interacción. A pesar del gran número de herramientas no exhaustivas capaces de encontrar las interacciones de forma consistente cuando estas presentan efectos marginales, solo las exhaustivas son capaces de hacer lo mismo cuando estos efectos desaparecen. Además, el estudio revela que la mayoría de métodos no tienen en consideración los falsos positivos, siendo *BEAM3* la única herramienta capaz de detectar epistasia a la vez que mantiene los errores de tipo I bajo control.

- **Una aplicación CPU paralela para la detección de epistasia de cualquier orden, empleando todos los niveles de paralelismo de un clústeres homogéneos de CPUs *x86_64*.** Esta implementación combina un algoritmo SIMD que implementa el test de asociación basado en la IM con una ejecución distribuida para aprovechar el paralelismo a nivel de bit, de datos y de tarea disponibles en un clúster de CPUs. El programa incluye implementaciones vectoriales explícitas para las extensiones vectoriales *AVX2* y *AVX512BW* de la arquitectura *x86_64*, así como código C++ estándar fácilmente vectorizable por un compilador para otras arquitecturas. Además, la herramienta implementa un algoritmo distribuido para la detección de epistasia de cualquier nivel que explota los recursos CPU de un clúster a través de MPI y multithreading. Esta herramienta es la más rápida y soporta más casuísticas para la búsqueda de epistasia que cualquier otra alter-

nativa del estado del arte que conocemos.

- **Un método innovador para el cálculo de tablas de penetrancia bajo modelos de epistasia de orden alto.** Los métodos existentes para el cálculo de tablas de penetrancia intentan resolver un sistema de ecuaciones muy complicado, resultante de sustituir en las ecuaciones de la prevalencia y heredabilidad las probabilidades de desarrollar el fenotipo condicionadas al genotipo del individuo por las expresiones bivariable del modelo de epistasia. Esta aproximación al problema puede producir sistemas de ecuaciones incompatibles, pues no todos los valores de prevalencia y heredabilidad se pueden obtener de forma simultánea. Nuestro método fija solo uno de estos dos parámetros y obtiene el valor máximo alcanzable para el otro, reduciendo el riesgo de formular un sistema incompatible. El resultado es un método capaz de obtener tablas de penetrancia de orden alto para valores más realistas de prevalencia y heredabilidad.

Estas contribuciones dieron lugar a tres programas diferentes: *MPI3SNP*, *Fiuncho* y *Toxo*. Tanto *MPI3SNP* como *Toxo* son distribuidos como código libre y usan CMake para su configuración y compilación. Los programas emplean una interfaz de línea de comandos, son ejecutados a través de MPI y la salida es proporcionada como un fichero de texto. Por el contrario, *Toxo* es una librería escrita en MATLAB que proporciona funciones para la lectura del modelo desde un fichero de entrada, el cálculo de la tabla de penetrancia a partir del modelo y una serie de parámetros, y una última función para escribir la tabla resultante en varios formatos de salida. Es dirigido a una audiencia con conocimientos de programación y su uso es más complejo que los dos primeros programas.

La investigación aquí presentada ofrece varios caminos por los que proseguir para continuar mejorando la detección de epistasia. Ambos programas solo son capaces de devolver las combinaciones con los valores más altos de IM encontrados. Aunque esta métrica es útil para comparar el grado de asociación entre combinaciones de variantes diferentes, es muy difícil juzgar si esta asociación es finalmente relevante usando exclusivamente el valor de IM. Una solución a este problema puede ser calcular un p-valor asociado a estas combinaciones que finalmente son reportadas. Yendo más allá, también podría ser factible distinguir entre efectos aditivos y no aditivos en el test de asociación, de forma similar a como Bayat y col. distinguen entre poder de asociación y efecto de interacción en [92].

Una línea de trabajo futuro diferente es la mejora del poder de detección de aproximaciones no exhaustivas. Los métodos actuales no son capaces de detectar epistasia ante la ausencia de efectos marginales. A pesar de los avances en la estrategia exhaustiva presentados en esta tesis, esta sigue sin ser viable para la búsqueda de epistasia de orden alto usando conjuntos de datos muy grandes. Por este motivo, es necesario continuar desarrollando nuevas estrategias no exhaustivas que mejoren el rendimiento en escenarios sin efectos marginales, e incluyendo técnicas HPC cuando sea necesario.

En cuanto a *Toxo* (y *PyToxo*) se refiere, el trabajo futuro incluye expandir los escenarios de uso soportados. La aproximación actual solo permite fijar la prevalencia o heredabilidad del fenotipo simulado mientras que el otro es maximizado. Sin embargo, una vez obtenido el máximo valor alcanzable de prevalencia o heredabilidad, es posible plantear un sistema de ecuaciones compatible siempre que el valor elegido sea menor que el máximo, y obtener una tabla de penetrancia para valores de prevalencia y heredabilidad fijados simultáneamente.

Los resultados de la investigación llevada a cabo a lo largo de esta tesis doctoral han sido publicados en las siguientes revistas internacionales:

- C. Ponte-Fernández, J. González-Domínguez y M. J. Martín. «Fiuncho: a program for any-order epistasis detection in CPU clusters». En: *The Journal of Supercomputing* (2022)
- B. González-Seoane, C. Ponte-Fernández, J. González-Domínguez y M. J. Martín. «PyToxo: a Python tool for calculating penetrance tables of high-order epistasis models». En: *BMC Bioinformatics* 23.117 (2022)
- C. Ponte-Fernández, J. González-Domínguez y M. J. Martín. «A SIMD algorithm for the detection of epistatic interactions of any order». En: *Future Generation Computer Systems* 132 (2022), págs. 108-123
- C. Ponte-Fernández, J. González-Domínguez y M. J. Martín. «Evaluation of existing methods for high-order epistasis detection». En: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.2 (2022), págs. 912-926
- C. Ponte-Fernández, J. González-Domínguez, A. Carvajal-Rodríguez y M. J. Martín. «Toxo: a library for calculating penetrance tables of high-order epistasis models». En: *BMC Bioinformatics* 21.138 (2020)

- C. Ponte-Fernández, J. González-Domínguez y M. J. Martín. «Fast search of third-order epistatic interactions on CPU and GPU clusters». En: *The International Journal of High Performance Computing Applications* 34.1 (2020), págs. 20-29

También se ha expuesto un breve resumen de los resultados alcanzados en esta tesis en la siguiente sesión de pósteres:

- C. Ponte-Fernández, J. González-Domínguez y M. J. Martín. «Poster: Genome wide association studies in high performance computing systems». En: *17th International Summer School on Advanced Computer Architecture and Compilation for High-performance Embedded Systems*. Fiuggi (Italy), 2021

Todos los programas y datasets aquí presentados se distribuyen como código libre, y se encuentran alojados en GitHub:

- *MPI3SNP*:

  https://github.com/UDC-GAC/mpi3snp

- *Fiuncho*:

  https://github.com/UDC-GAC/fiuncho

- *Toxo*:

  https://github.com/UDC-GAC/toxo

- Datasets used in Chapter 3:

  https://github.com/UDC-GAC/epistasis-simulation-data

Bibliography

[1] E. Génin. «Missing heritability of complex diseases: case solved?» In: *Human Genetics* 139 (2020), pp. 103–113.

[2] C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Fast search of third-order epistatic interactions on CPU and GPU clusters». In: *The International Journal of High Performance Computing Applications* 34.1 (2020), pp. 20–29.

[3] C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Evaluation of existing methods for high-order epistasis detection». In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.2 (2022), pp. 912–926.

[4] C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «A SIMD algorithm for the detection of epistatic interactions of any order». In: *Future Generation Computer Systems* 132 (2022), pp. 108–123.

[5] C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Fiuncho: a program for any-order epistasis detection in CPU clusters». In: *The Journal of Supercomputing* (2022).

[6] C. Ponte-Fernández, J. González-Domínguez, A. Carvajal-Rodríguez, and M. J. Martín. «Toxo: a library for calculating penetrance tables of high-order epistasis models». In: *BMC Bioinformatics* 21.138 (2020).

[7] B. González-Seoane, C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «PyToxo: a Python tool for calculating penetrance tables of high-order epistasis models». In: *BMC Bioinformatics* 23.117 (2022).

[8] C. Ponte-Fernández, J. González-Domínguez, and M. J. Martín. «Poster: Genome wide association studies in high performance computing systems». In: *17th International Summer School on Advanced Computer Architecture and Compilation for High-performance Embedded Systems*. Fiuggi (Italy), 2021.

[9] S. Fairley, E. Lowy-Gallego, E. Perry, and P. Flicek. «The International Genome Sample Resource (IGSR) collection of open human genomic variation resources». In: *Nucleic Acids Research* 48.D1 (Oct. 2019), pp. D941–D947.

[10] S. Das, L. Forer, S. Schönherr, C. Sidore, A. E. Locke, A. Kwong, S. I. Vrieze, E. Y. Chew, S. Levy, M. McGue, et al. «Next-generation genotype imputation service and methods». In: *Nature genetics* 48.10 (2016), pp. 1284–1287.

[11] E. Uffelmann, Q. Q. Huang, N. S. Munung, J. de Vries, Y. Okada, A. R. Martin, H. C. Martin, T. Lappalainen, and D. Posthuma. «Genome-wide association studies». In: *Nature Reviews Methods Primers* 1.59 (2021).

[12] E. Zeggini and J. P. Ioannidis. «Meta-analysis in genome-wide association studies». In: *Pharmacogenomics* 10.2 (2009), pp. 191–201.

[13] P. Kraft, E. Zeggini, and J. P. Ioannidis. «Replication in genome-wide association studies». In: *Statistical science: a review journal of the Institute of Mathematical Statistics* 24.4 (2009), p. 561.

[14] A. Buniello, J. A. L. MacArthur, M. Cerezo, L. W. Harris, J. Hayhurst, C. Malangone, A. McMahon, J. Morales, E. Mountjoy, E. Sollis, et al. «The NHGRI-EBI GWAS Catalog of published genome-wide association studies, targeted arrays and summary statistics 2019». In: *Nucleic acids research* 47.D1 (2019), pp. D1005–D1012.

[15] L. A. Hindorff, P. Sethupathy, H. A. Junkins, E. M. Ramos, J. P. Mehta, F. S. Collins, and T. A. Manolio. «Potential etiologic and functional implications of genome-wide association loci for human diseases and traits». In: *Proceedings of the National Academy of Sciences* 106.23 (2009), pp. 9362–9367.

[16] L. W. Hahn, M. D. Ritchie, and J. H. Moore. «Multifactor dimensionality reduction software for detecting gene–gene and gene–environment interactions». In: *Bioinformatics* 19.3 (2003), pp. 376–382.

[17] R. Campos, D. Marques, S. Santander-Jiménez, L. Sousa, and A. Ilic. «Heterogeneous CPU+iGPU processing for efficient epistasis detection». In: *European Conference on Parallel Processing*. Springer, 2020, pp. 613–628.

[18] L. Wienbrandt, J. C. Kässens, and D. Ellinghaus. «SNPInt-GPU: tool for epistasis testing with multiple methods and GPU acceleration». In: *Epistasis: methods and protocols*. Springer US, 2021, pp. 17–35.

[19] R. Nobre, A. Ilic, S. Santander-Jiménez, and L. Sousa. «Exploring the binary precision capabilities of tensor cores for epistasis detection». In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 338–347.

[20] J. González-Domínguez, L. Wienbrandt, J. C. Kässens, D. Ellinghaus, M. Schimmler, and B. Schmidt. «Parallelizing epistasis detection in GWAS on FPGA and GPU-accelerated computing systems». In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 12.5 (2015), pp. 982–994.

[21] J. C. Kässens, L. Wienbrandt, J. González-Domínguez, B. Schmidt, and M. Schimmler. «High-speed exhaustive 3-locus interaction epistasis analysis on FPGAs». In: *Journal of Computational Science* 9 (2015), pp. 131–136.

[22] G. Ribeiro, N. Neves, S. Santander-Jiménez, and A. Ilic. «HEDAcc: FPGA-based accelerator for high-order epistasis detection». In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 124–132.

[23] R. Nobre, A. Ilic, S. Santander-Jiménez, and L. Sousa. «Fourth-order exhaustive epistasis detection for the xPU era». In: *50th International Conference on Parallel Processing*. 27. 2021.

[24] R. Nobre, S. Santander-Jiménez, L. Sousa, and A. Ilic. «Accelerating 3-way epistasis detection with CPU+GPU processing». In: *Workshop on job scheduling strategies for parallel processing*. Springer, 2020, pp. 106–126.

[25] L. Wienbrandt, J. C. Kässens, M. Hübenthal, and D. Ellinghaus. «1000× faster than PLINK: combined FPGA and GPU accelerators for logistic regression-based detection of epistasis». In: *Journal of Computational Science* 30 (2019), pp. 183–193.

[26] B. Goudey, M. Abedini, J. L. Hopper, M. Inouye, E. Makalic, D. F. Schmidt, J. Wagner, Z. Zhou, J. Zobel, and M. Reumann. «High performance computing enabling exhaustive analysis of higher order single nucleotide polymorphism interaction in genome wide association studies». In: *Health information science and systems* 3.S3 (2015).

[27] A. Upton, O. Trelles, J. A. Cornejo-García, and J. R. Perkins. «Review: high-performance computing to detect epistasis in genome scale data sets». In: *Briefings in Bioinformatics* 17.3 (2016), pp. 368–379.

[28] J. H. Moore. «Mining patterns of epistasis in human genetics». In: *Biological Data Mining*. Chapman and Hall/CRC, 2009, pp. 207–224.

[29] J. Shang, J. Zhang, Y. Sun, and Y. Zhang. «EpiMiner: a three-stage co-information based method for detecting and visualizing epistatic interactions». In: *Digital Signal Processing* 24 (2014), pp. 1–13.

[30] D. Jünger, C. Hundt, J. G. Domínguez, and B. Schmidt. «Speed and accuracy improvement of higher-order epistasis detection on CUDA-enabled GPUs». In: *Cluster Computing* 20.3 (2017), pp. 1899–1908.

[31] S. D. Turner, S. M. Dudek, and M. D. Ritchie. «ATHENA: A knowledge-based hybrid backpropagation-grammatical evolution neural network algorithm for discovering epistasis among quantitative trait loci». In: *BioData Mining* 3.5 (2010).

[32] P.-J. Jing and H.-B. Shen. «MACOED: a multi-objective ant colony optimization algorithm for SNP epistasis detection in genome-wide association studies». In: *Bioinformatics* 31.5 (2015), pp. 634–641.

[33] M. Y. Park and T. Hastie. «Penalized logistic regression for detecting gene interactions». In: *Biostatistics* 9.1 (2008), pp. 30–50.

[34] M. D. Ritchie. «Using biological knowledge to uncover the mystery in the search for epistasis in genome-wide association studies». In: *Annals of human genetics* 75.1 (2011), pp. 172–182.

[35] W. S. Bush, S. M. Dudek, and M. D. Ritchie. «Biofilter: a knowledge-integration system for the multi-locus analysis of genome-wide association studies». In: *Biocomputing 2009*. World Scientific, 2009, pp. 368–379.

[36] C. Herold, M. Steffens, F. F. Brockschmidt, M. P. Baur, and T. Becker. «INTERSNP: genome-wide interaction analysis guided by a priori information». In: *Bioinformatics* 25.24 (2009), pp. 3275–3281.

[37] X. Guo, Y. Meng, N. Yu, and Y. Pan. «Cloud computing for detecting high-order genome-wide epistatic interaction via dynamic clustering». In: *BMC Bioinformatics* 15.102 (2014).

[38] S. Leem, H.-H. Jeong, J. Lee, K. Wee, and K.-A. Sohn. «Fast detection of high-order epistatic interactions in genome-wide association studies using information theoretic measure». In: *Computational Biology and Chemistry* 50 (2014), pp. 19–28.

[39] X. Wan, C. Yang, Q. Yang, H. Xue, X. Fan, N. L. Tang, and W. Yu. «BOOST: a fast approach to detecting gene-gene interactions in genome-wide case-control studies». In: *The American Journal of Human Genetics* 87.3 (2010), pp. 325–340.

[40] J. González-Domínguez and B. Schmidt. «GPU-accelerated exhaustive search for third-order epistatic interactions in case-control studies». In: *The Journal of Computational Science* 8 (2015), pp. 93–100.

[41] Intel Corporation. *Intel Xeon processor E5 v3 product family specification update*. Last accessed: March 16th, 2022. 2017. URL: https://cdrdv2.intel.com/v1/dl/getContent/330785.

[42] J. Shang, X. Wang, X. Wu, Y. Sun, Q. Ding, J.-X. Liu, and H. Zhang. «A review of ant colony optimization based methods for detecting epistatic interactions». In: *IEEE Access* 7 (2019), pp. 13497–13509.

[43] S. Uppu, A. Krishna, and R. P. Gopalan. «A review on methods for detecting SNP interactions in high-dimensional genomic data». In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 15.2 (2018), pp. 599–612.

[44] X. Ding and X. Guo. «A survey of SNP data analysis». In: *Big Data Mining and Analytics* 1.3 (2018), pp. 173–190.

[45] C. Niel, C. Sinoquet, C. Dina, and G. Rocheleau. «A survey about methods dedicated to epistasis detection». In: *Frontiers in genetics* 6.285 (2015).

[46] W.-H. Wei, G. Hemani, and C. S. Haley. «Detecting epistasis in human complex traits». In: *Nature Reviews Genetics* 15.11 (2014), pp. 722–733.

[47] C. L. Koo, M. J. Liew, M. S. Mohamad, and A. H. Mohamed Salleh. «A review for detecting gene-gene interactions using machine learning methods in genetic epidemiology». In: *BioMed Research International* 2013.432375 (2013).

[48] S. Tuo, H. Chen, and H. Liu. «A survey on swarm intelligence search methods dedicated to detection of high-order SNP interactions». In: *IEEE Access* 7 (2019), pp. 162229–162244.

[49] C. Chatelain, G. Durand, V. Thuillier, and F. Augé. «Performance of epistasis detection methods in semi-simulated GWAS». In: *BMC Bioinformatics* 19.231 (2018).

[50] J. Shang, J. Zhang, Y. Sun, D. Liu, D. Ye, and Y. Yin. «Performance analysis of novel methods for detecting epistasis». In: *BMC Bioinformatics* 12.475 (2011).

[51] Y. Wang, G. Liu, M. Feng, and L. Wong. «An empirical comparison of several recent epistatic interaction detection methods». In: *Bioinformatics* 27.21 (2011), pp. 2936–2943.

[52] C. C. M. Chen, H. Schwender, J. Keith, R. Nunkesser, K. Mengersen, and P. Macrossan. «Methods for identifying SNP interactions: a review on variations of logic regression, random forest and bayesian logistic regression». In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 8.6 (2011), pp. 1580–1591.

[53] M. D. Ritchie and K. V. Steen. «The search for gene-gene interactions in genome-wide association studies: challenges in abundance of methods, practical considerations, and biological interpretation». In: *Annals of Translational Medicine* 6.8 (2018), p. 157.

[54] M. D. Ritchie. «Finding the epistasis needles in the genome-wide haystack». In: *Epistasis: methods and protocols*. Springer New York, 2015, pp. 19–33.

[55] X. Sun, Q. Lu, S. Mukherjee, P. K. Crane, R. Elston, and M. D. Ritchie. «Analysis pipeline for the epistasis search – statistical versus biological filtering». In: *Frontiers in Genetics* 5.106 (2014).

[56] J. Shang, J. Zhang, X. Lei, Y. Zhang, and B. Chen. «Incorporating heuristic information into ant colony optimization for epistasis detection». In: *Genes & Genomics* 34.3 (2012), pp. 321–327.

[57] R. T. Guy, P. Santago, and C. D. Langefeld. «Bootstrap aggregating of alternating decision trees to detect sets of SNPs that associate with disease». In: *Genetic Epidemiology* 36.2 (2012), pp. 99–106.

[58] Y. Zhang. «A novel bayesian graphical model for genome-wide multi-SNP association mapping». In: *Genetic Epidemiology* 36.1 (2012), pp. 36–47.

[59] J. Wang, T. Joshi, B. Valliyodan, H. Shi, Y. Liang, H. T. Nguyen, J. Zhang, and D. Xu. «A bayesian model for detection of high-order interactions among genetic variants in genome-wide association studies». In: *BMC Genomics* 16.1011 (2015).

[60] J. Shang, Y. Sun, J.-X. Liu, J. Xia, J. Zhang, and C.-H. Zheng. «CINOEDV: a co-information based method for detecting and visualizing n-order epistatic interactions». In: *BMC Bioinformatics* 17.214 (2016).

[61] Y. Sun, X. Wang, J. Shang, J. Liu, C. Zheng, and X. Lei. «Introducing heuristic information into ant colony optimization algorithm for identifying epistasis». In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 17.4 (2018), pp. 1253–1261.

[62] M. Xie, J. Li, and T. Jiang. «Detecting genome-wide epistases based on the clustering of relatively frequent items». In: *Bioinformatics* 28.1 (2012), pp. 5–12.

[63] Y. Sun, J. Shang, J.-X. Liu, S. Li, and C.-H. Zheng. «epiACO - a method for identifying epistasis based on ant colony optimization algorithm». In: *BioData Mining* 10.23 (2017).

[64] S. Tuo. «FDHE-IW: a fast approach for detecting high-order epistasis in genome-wide case-control studies». In: *Genes* 9.9 (2018), p. 435.

[65] R. J. Urbanowicz and J. H. Moore. «The application of Pittsburgh-Style learning classifier systems to address genetic heterogeneity and epistasis in association studies». In: *Parallel problem solving from nature, PPSN XI*. Springer Berlin Heidelberg, 2010, pp. 404–413.

[66] J. Liu, G. Yu, Y. Jiang, and J. Wang. «HiSeeker: detecting high-order SNP interactions based on pairwise SNP combinations». In: *Genes* 8.6 (2017), p. 153.

[67] Y. Sun, J. Shang, J. Liu, and S. Li. «An improved ant colony optimization algorithm for the detection of SNP-SNP interactions». In: *Intelligent computing methodologies*. Springer International Publishing, 2016, pp. 21–32.

[68] A. Terada, R. Yamada, K. Tsuda, and J. Sese. «LAMPLINK: detection of statistically significant SNP combinations from GWAS data». In: *Bioinformatics* 32.22 (2016), pp. 3513–3515.

[69] C. Wei and Q. Lu. «GWGGI: software for genome-wide gene-gene interaction analysis». In: *BMC Genetics* 15.101 (2014).

[70] M. D. Ritchie, L. W. Hahn, N. Roodi, L. R. Bailey, W. D. Dupont, F. F. Parl, and J. H. Moore. «Multifactor-dimensionality reduction reveals high-order interactions among estrogen-metabolism genes in sporadic breast cancer». In: *American Journal of Human Genetics* 69.1 (2001), pp. 138–147.

[71] D. J. Miller, Y. Zhang, G. Yu, Y. Liu, L. Chen, C. D. Langefeld, D. Herrington, and Y. Wang. «An algorithm for learning maximum entropy probability models of disease risk that efficiently searches and sparingly encodes multilocus genomic interactions». In: *Bioinformatics* 25.19 (2009), pp. 2478–2485.

[72] T. T. Wu, Y. F. Chen, T. Hastie, E. Sobel, and K. Lange. «Genome-wide association analysis by lasso penalized logistic regression». In: *Bioinformatics* 25.6 (2009), pp. 714–721.

[73] S. Tuo, J. Zhang, X. Yuan, Z. He, Y. Liu, and Z. Liu. «Niche harmony search algorithm for detecting complex disease associated high-order SNP combinations». In: *Scientific Reports* 7.11529 (2017).

[74] C. Yang, Z. He, X. Wan, Q. Yang, H. Xue, and W. Yu. «SNPHarvester: a filtering-based approach for detecting epistatic interactions in genome-wide association studies». In: *Bioinformatics* 25.4 (2009), pp. 504–511.

[75] X. Wan, C. Yang, Q. Yang, H. Xue, N. L. Tang, and W. Yu. «Predictive rule inference for epistatic interaction detection in genome-wide association studies». In: *Bioinformatics* 26.1 (2010), pp. 30–37.

[76] T. Uno, M. Kiyomi, and H. Arimura. «LCM ver. 2: efficient mining algorithms for frequent/closed/maximal itemsets». In: *FIMI '04, Proceedings of the IEEE ICDM workshop on frequent itemset mining implementations*. Vol. 126. Brighton, UK, 2004.

[77] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr. «A survey on metaheuristics for stochastic combinatorial optimization». In: *Natural Computing* 8.2 (2009), pp. 239–287.

[78] J. Kennedy. «Swarm intelligence». In: *Handbook of nature-inspired and innovative computing: integrating classical models with emerging technologies*. Springer US, 2006, pp. 187–219.

[79] X.-S. Yang. «Harmony search as a metaheuristic algorithm». In: *Music-inspired harmony search algorithm: theory and applications*. Vol. 191. Springer Berlin Heidelberg, 2009.

[80] D. Whitley. «A genetic algorithm tutorial». In: *Statistics and Computing* 4.2 (1994), pp. 65–85.

[81] X. Llorà and J. M. Garrell. «Knowledge-independent data mining with fine-grained parallel evolutionary algorithms». In: *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 461–468.

[82] M. O'Neill and C. Ryan. «Grammatical evolution». In: *IEEE Transactions on Evolutionary Computation* 5.4 (2001), pp. 349–358.

[83] R. J. Urbanowicz, J. Kiralis, N. A. Sinnott-Armstrong, T. Heberling, J. M. Fisher, and J. H. Moore. «GAMETES: a fast, direct algorithm for generating pure, strict, epistatic models with random architectures». In: *BioData Mining* 5.16 (2012).

[84] J. Marchini, P. Donnelly, and L. R. Cardon. «Genome-wide strategies for detecting multiple loci that influence complex diseases». In: *Nature Genetics* 37.4 (2005), pp. 413–417.

[85] J.-H. Park, M. H. Gail, C. R. Weinberg, R. J. Carroll, C. C. Chung, Z. Wang, S. J. Chanock, J. F. Fraumeni, and N. Chatterjee. «Distribution of allele frequencies and effect sizes and their interrelationships for common genetic susceptibility variants». In: *Proceedings of the National Academy of Sciences* 108.44 (2011), pp. 18026–18031.

[86] R. Walters, C. Laurin, and G. H. Lubke. «An integrated approach to reduce the impact of minor allele frequency and linkage disequilibrium on variable importance measures for genome-wide data». In: *Bioinformatics* 28.20 (2012), pp. 2615–2623.

[87] A. Caballero. *Quantitative genetics*. Cambridge University Press, 2020.

[88] T. J. Polderman, B. Benyamin, C. A. De Leeuw, P. F. Sullivan, A. Van Bochoven, P. M. Visscher, and D. Posthuma. «Meta-analysis of the heritability of human traits based on fifty years of twin studies». In: *Nature genetics* 47 (2015), pp. 702–709.

[89] S. Harari. «Why we should care about ultra-rare disease». In: *European Respiratory Review* 25.140 (2016), pp. 101–103.

[90] W. Muła, N. Kurz, and D. Lemire. «Faster population counts using AVX2 instructions». In: *The Computer Journal* 61.1 (2018), pp. 111–120.

[91] Intel Corporation. *2nd gen. Intel Xeon Scalable Processors specification update*. Last accessed: March 16th, 2022. 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/338848.

[92] A. Bayat, B. Hosking, Y. Jain, C. Hosking, M. Kodikara, D. Reti, N. A. Twine, and D. C. Bauer. «Fast and accurate exhaustive higher-order epistasis search with BitEpi». In: *Scientific reports* 11.15923 (2021).

[93] B. Peng and M. Kimmel. «simuPOP: a forward-time population genetics simulation environment». In: *Bioinformatics* 21.18 (2005), pp. 3686–3687.

[94] F. A. Wright, H. Huang, X. Guan, K. Gamiel, C. Jeffries, W. T. Barry, F. Pardo-Manuel de Villena, P. F. Sullivan, K. C. Wilhelmsen, and F. Zou. «Simulating association studies: a data-based resampling method for candidate regions or whole genome scans». In: *Bioinformatics* 23.19 (2007), pp. 2581–2588.

[95] M. Pérez-Enciso, N. Forneris, G. d. l. Campos, and A. Legarra. «Evaluating sequence-based genomic prediction with an efficient new simulator». In: *Genetics* 205.2 (2017), pp. 939–953.

[96] Y. Zhang and J. S. Liu. «Bayesian inference of epistatic interactions in case-control studies». In: *Nature Genetics* 39.9 (2007), pp. 1167–1173.

[97] T. L. Edwards, W. S. Bush, S. D. Turner, S. M. Dudek, E. S. Torstenson, M. Schmidt, E. Martin, and M. D. Ritchie. «Generating linkage disequilibrium patterns in data simulations using genomeSIMLA». In: *Evolutionary computation, machine learning and data mining in bioinformatics*. Springer Berlin Heidelberg, 2008, pp. 24–35.

[98] J. Shang, J. Zhang, X. Lei, W. Zhao, and Y. Dong. «EpiSIM: simulation of multiple epistasis, linkage disequilibrium patterns and haplotype blocks for genome-wide interaction analysis». In: *Genes & Genomics* 35.3 (2013), pp. 305–16.

[99] J. Li and Y. Chen. «Generating samples for association studies based on HapMap data». In: *BMC Bioinformatics* 9.44 (2008).

[100] R. J. Neuman, J. P. Rice, and A. Chakravarti. «Two-locus models of disease». In: *Genetic Epidemiology* 9.5 (1992), pp. 347–365.

[101] The MathWorks, Inc. *MATLAB Math Symbolic Toolbox*. Natick, Massachusetts, United States, 2018.

[102] C. Niel, C. Sinoquet, C. Dina, and G. Rocheleau. «SMMB: a stochastic Markov blanket framework strategy for epistasis detection in GWAS». In: *Bioinformatics* 34.16 (2018), pp. 2773–2780.

[103] X. Cao, G. Yu, J. Liu, L. Jia, and J. Wang. «ClusterMI: detecting high-order SNP interactions based on clustering and mutual information». In: *International Journal of Molecular Sciences* 19.8 (2018), p. 2267.

[104] C.-H. Yang, L.-Y. Chuang, and Y.-D. Lin. «Multiobjective multifactor dimensionality reduction to detect SNP–SNP interactions». In: *Bioinformatics* 34.13 (2018), pp. 2228–2236.

[105] P. C. Phillips. «Epistasis — the essential role of gene interactions in the structure and evolution of genetic systems». In: *Nature Reviews Genetics* 9 (2008), pp. 855–867.

[106] D. Speed, N. Cai, the UCLEB Consortium, M. R. Johnson, S. Nejentsev, and D. J. Balding. «Reevaluation of SNP heritability in complex human traits». In: *Nature Genetics* 49.7 (2017), pp. 986–992.

[107] F. C. G. Polubriaginof, R. Vanguri, K. Quinnies, G. M. Belbin, A. Yahi, H. Salmasian, T. Lorberbaum, V. Nwankwo, L. Li, M. M. Shervey, P. Glowe, I. Ionita-Laza, M. Simmerling, G. Hripcsak, S. Bakken, D. Goldstein, K. Kiryluk, E. E. Kenny, J. Dudley, D. K. Vawdrey, and N. P. Tatonetti. «Disease heritability inferred from familial relationships reported in medical records». In: *Cell* 173.7 (2018), 1692–1704.e11.

[108] B. Han, M. Park, and X.-w. Chen. «A Markov blanket-based method for detecting causal SNPs in GWAS». In: *BMC Bioinformatics* 11.S5 (2010).