UNIVERSITÉ DE GENÈVE

Archive ouverte UNIGE

https://archive-ouverte.unige.ch

Thèse 2016

Open Access

This version of the publication is provided by the author(s) and made available in accordance with the copyright holder(s).

Breaking computational barriers : application of computational science on high performance computers

Meyer, Xavier

How to cite

MEYER, Xavier. Breaking computational barriers : application of computational science on high performance computers. Doctoral Thesis, 2016. doi: 10.13097/archive-ouverte/unige:86518

This publication URL: https://archive-ouverte.unige.ch/unige:86518
Publication DOI: 10.13097/archive-ouverte/unige:86518

© This document is protected by copyright. Please refer to copyright holder(s) for terms of use.

Université de Genève
Département d'Informatique

Faculté des sciences
Professeur Bastien Chopard

Université de Lausanne
Département d'Ecologie et d'Evolution

Faculté de biologie et médecine
Professeur Nicolas Salamin

Haute école du paysage d'ingénierie et d'architecture
Département d' Ingénierie des Technologies de l'Information
Professeur Paul Albuquerque

Breaking Computational Barriers

Application of computational science on high performance computers

THÈSE

présentée à la Faculté des Sciences de l'Université de Genève pour obtenir le grade de Docteur ès Sciences, mention informatique

par

Xavier Meyer

de La Tour-de-Peilz (VD, Suisse)

Thèse N°4966

Genève
Atelier d'impression Uni-Mail
2016

UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

Doctorat ès sciences
Mention informatique

Thèse de *Monsieur Xavier MEYER*

intitulée :

"Breaking Computational Barriers

Application of Computational Science on High Performance Computers"

La Faculté des sciences, sur le préavis de Monsieur B. CHOPARD, professeur ordinaire et directeur de thèse (Département d'informatique), Monsieur N. SALAMIN, professeur et codirecteur de thèse (Université de Lausanne, Département d'écologie et évolution), Monsieur P. ALBUQUERQUE, professeur et codirecteur de thèse (Haute École du paysage, d'ingénierie et d'architecture de Genève, Département d'ingénierie des technologies de l'information), Monsieur J.-L. FALCONE, docteur (Département d'informatique), Monsieur B. BOUSSAU, docteur (Laboratoire de biométrie et biologie évolutive, Université Claude Bernard Lyon 1, France) autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 21 juillet 2016

Thèse - 4966 -

Le Doyen

N.B. - La thèse doit porter la déclaration précédente et remplir les conditions énumérées dans les "Informations relatives aux thèses de doctorat à l'Université de Genève".

Remerciements

Le travail présenté dans ce manuscrit peut s'apparenter à une longue aventure parsemée de nombreuses difficultés qu'il m'a été possible de surmonter grâce à la présence et au soutien de nombreuses personnes. Il est donc naturel de débuter le récit de cette aventure par l'expression de ma gratitude envers tous ceux sur qui j'ai pu compter, à commencer par mes « guides ».

Mes remerciements vont tout d'abord à mon directeur de thèse, le Prof. Bastien Chopard, de m'avoir donné l'opportunité de rejoindre son groupe de recherche ainsi que la confiance qu'il m'a octroyé tout au long de cette thèse. Je me dois aussi d'exprimer toute ma reconnaissance à mes deux co-directeurs de thèse pour leur disponibilité et soutien. Merci donc au Prof. Paul Albuquerque de m'avoir initié au calcul parallèle durant mes études d'ingénierie, de m'avoir légué sa cave et finalement de m'avoir accompagné dans mes premiers pas de thésard. Mes sincères remerciements vont au Prof. Nicolas Salamin d'avoir pris le pari d'accueillir l'ignare en biologie que j'étais dans son groupe de phylogénétique computationnelle, d'avoir inlassablement témoigné de l'intérêt à mes pérégrinations et de m'avoir fait grandir en tant que chercheur. Finalement, bien qu'il n'ait pas directement participé à cette thèse, je me dois de remercier mon mentor, le chef et Prof. Michel Lazeyras, de m'avoir permis de goûter à l'univers de la recherche académique et de m'avoir encouragé à poursuivre cette voie.

Toute ma reconnaissance va au deux experts de mon jury qui m'ont fait l'honneur d'accepter de participer à mon Jury et donc de statuer sur le fruit de cette longue aventure. Merci au Dr. Jean-Luc Falcone d'avoir accepter cette invitation malgré un si court délai et au Dr. Bastien Boussau pour sa minutieuse et attentive lecture du manuscrit ainsi que ses nombreuses questions constructives.

Cette aventure a aussi été animée par de nombreux compagnons d'infortune qui ont su égayer mon quotidien et alléger mon fardeau dans les moments difficiles. En effet, qu'aurait été cette aventure sans ma première famille académique : le groupe SPC! Je ne peux que commencer par chaleureusement remercier Orestis et Christophe qui ont égayés ma thèse dès ses premiers instants, jusqu'à son aboutissement : merci pour votre soutien et les inoubliables souvenirs. Merci également aux habitants initiaux de mon premier bureau, Andrea, Daniel et Kali, pour votre accueil chaleureux et merci aussi à tous les autres membres du groupe à savoir, Aziza, Federico, Gregor, Jonas, Léa, Mohamed, Pierre, Reto, Sébastien, Sha Li et Yann pour tous ces bons moments passés autour d'un repas ou d'un café.

Chanceux que je suis, j'ai aussi eu le privilège d'être accueilli à bras ouverts dans le groupe de phylogénétique computationnelle de l'université de Lausanne. J'ai grandement apprécié m'initier à la biologie évolutive au sein de ce groupe où d'innombrables discussions enrichissantes ont occupé pauses café, lunch et réunions de groupe. Un énorme

merci à Linda et Daniele pour les échanges scientifiques ainsi que pour leur gentillesse, à Martha pour avoir réveillé mon « quart colombien », à Kana pour les histoires matinales ainsi qu'à Anna (x2), Glenn, Guangpeng, Iakov, Jonathan, Oriane, Sacha et Talita, pour votre soutien, l'éternelle bonne ambiance que vous avez su faire vivre et les bons moments partagés.

Lorsqu'il m'est arrivé de réussir à m'éloigner un peu de cette aventure académique, j'ai toujours pu compter sur la présence et l'écoute de ma famille et de mes amis. Un énorme merci à Arnaud, Filipe, Yann S. et Yorick et les potes du jeudredi pour avoir su me changer les idées. Et finalement, toute ma gratitude et mon affection vont à mes parents, Astrid et Jacques, mon frère, Thiéry, mon grand-père Max, mon beau-père et ami, Alexandre et ma copine, Sarah ainsi qu'à ses proches. Votre compréhension et vos encouragements ont été une source intarissable de motivation durant ces dernières années : Merci !

Résumé

Consulter des prévisions météorologiques ou déterminer le meilleur itinéraire pour un trajet est de nos jours devenu une banalité. Ces gestes du quotidien requièrent néanmoins de coûteux et complexes calculs numériques qui sont aujourd'hui possible grâce aux progrès des sciences numériques. Ces progrès ne visent pas uniquement à simplifier notre quotidien, ils permettent d'appréhender les défis scientifiques majeurs de notre époque tels que l'étude des collisions de particules aux CERN ou l'acquisition du génome humain en son entier.

Afin de résoudre ces défis de la science moderne, de nombreux concepts avancés issus d'années de recherche en informatique et mathématique appliqué ont été employés. En effet, les calculs résultant d'une modélisation méticuleuse des phénomènes étudiés ont été rendus possible grâce aux nombreuses méthodes numériques développées tout au long du 20ème siècle. Ces méthodes, depuis leur conception initiale, n'ont eu de cesse d'être améliorées, tout comme leurs implémentations logicielles. Additionnellement à ces améliorations, la rapide évolution du matériel informatique a permis l'investigation de phénomènes de complexité croissante.

Les sciences numériques ont donc permis des projets scientifiques sans précédent à être conduit en rendant possible le calcul rapide de volume de données que personne ne pourrait imaginer traiter en une vie. Les progrès algorithmiques, méthodologiques et matériels ont non seulement élargi le champ d'investigations scientifiques envisageables en réduisant leur temps de calculs d'années en minutes, mais aussi permis à des scientifiques de tout horizons d'étudier des phénomènes trop complexes, petits, vastes ou tout simplement non-reproductibles dans un laboratoire.

Le futur des sciences numériques fait face à de sérieux défis. La limite de la miniaturisation des composants informatiques a mis un frein au flot jusqu'alors intarissable de gain de performance provenant du matériel informatique. Paradoxalement, le coût computationnel des défis scientifique du futur ne cesse de croître dû à l'émergence du « Big Data » ainsi que de l'intérêt croissant pour l'étude de phénomènes ayant plusieurs échelles de temps ou d'espace, voir les deux. Ce besoin accru de puissance computationnelle est tel que le calcul de haute performance exploitant des architectures de calcul massivement parallèle, tel que les supercalculateur ou les unités de calcul graphique, n'a jamais été aussi important qu'aujourd'hui.

Dans cette thèse, nous participons à ce développement en proposant des approches de calculs parallèle pour trois différents défis de notre époque. Dans la première partie nous étudions le potentiel de l'utilisation des unités de calcul graphique avec pour objectif d'accélérer un des algorithme les plus important du 20ème siècle : l'algorithme du simplexe. Dans une seconde partie, nous investiguons si le calcul hautement parallèle de dynamique des fluides peut élargir notre compréhension de l'impénétrable organe de

l'ouïe qui demeure de nos jours un mystère insondable. Dans la troisième et dernière partie, nous nous attelons aux problèmes calculatoires posés par l'étude de la théorie de l'évolution de Darwin. L'étude de l'évolution est plus que jamais d'actualité dû à la disponibilité croissante de données moléculaires qui, lorsque couplées à des modèles statistiques sophistiqués, représentent un défi computationnel jusqu'aujourd'hui insurmonté.

Abstract

Computational science is the application of computing capabilities to obtain solution to problems of the real world. It is omnipresent in our everyday life: for instance, it provides us weather forecast through simulation of meteorological phenomena or helps us to find the best itinerary for our trips by determining the solution to complex optimisation problems. More importantly, computational science makes possible today's scientific challenges, such as the study of colliding particles at CERN or the acquisition of the whole human genome, to be faced.

These crucial scientific and engineering problems are tackled using advanced concepts from computer science. Indeed, the calculations underlying to the mathematical models expressing these complex problems are realisable thanks to methods developed during the 20th century such as the Monte Carlo method or the Simplex algorithm. Through years, these methods, as well as their software implementations, were subject to a steady research effort that, jointly with the ever evolving computer architectures, enabled more and more complex problems to be investigated.

Computational science permitted thus unprecedented scientific projects to be conducted by enabling fast calculations on amounts of data that no person could apprehends in a lifetime. Better algorithms, implementations and hardware helped to broaden the range of investigations practicable by reducing their computing time from years to hours or minutes. However, computational science is not only about accelerating computations: it is also about enabling researchers from multiple horizons to investigate phenomena that are too complex, small, vast or simply not reproducible in laboratory experiments.

The future of computational science is facing serious challenges. Until recently, the computing power of processors was quickly increasing and, consequently, existing software was implicitly accelerated. However, physical limitations in the miniaturisation of computer chips halted this steady source of performance gains. Paradoxically, the computational cost of problems awaiting to be solved never stops to augment. Indeed, the emergence of *Big Data* in several fields are soliciting an even greater computing power and so are the models that encompass multiple scales of time, space or both.

Breaking these computational barriers is therefore a central concern that emphasises the crucial need for parallel and distributed computing in the future of computational science. Indeed, parallel algorithms and software designed for high performance computing clusters or novel massively-parallel architecture, such as graphical processing units, has never been so important.

In this thesis, we take part to this development by proposing parallel computing approaches for three different challenging problems. In a first part, we study the potential of using graphical processing units to accelerate one of the most important algorithm of the 20th century: the Simplex algorithm. In a second part, we investigate if the

use of highly-parallel computational fluid dynamics could broaden our understanding of the mechanism of the impenetrable organ of hearing. In the third and final part, we tackle the computational challenges raised by the study of Darwin's theory of evolution that stem from the use of computationally demanding statistical models and methods coupled with the ever growing amount of available molecular data.

Contents

I. Mixed Integer Programming on GPUs: A Case Study 1

1. Introduction 5
- 1.1. State of the art 5
- 1.2. Simplex algorithm 6
  - 1.2.1. Linear programming model 6
  - 1.2.2. Standard simplex algorithm 7
  - 1.2.3. Revised simplex method 8
  - 1.2.4. Heuristics and improvements 10
- 1.3. Branch-and-bound algorithm 12
  - 1.3.1. Integer linear programming 12
  - 1.3.2. Branch-and-bound tree 12
  - 1.3.3. Branching strategy 14
  - 1.3.4. Node selection strategy 14
  - 1.3.5. Cutting-plane methods 15
- 1.4. CUDA considerations 15
  - 1.4.1. Parallel reduction 16
  - 1.4.2. Kernel optimization 16

2. Implementations 17
- 2.1. Standard simplex 17
- 2.2. Revised simplex 19
- 2.3. Branch-and-bound 23

3. Performance model 25
- 3.1. Levels of parallelism 25
- 3.2. Amount of work done by a thread 26
- 3.3. Global performance model 26
- 3.4. A kernel example: *steepest-edge* 27
- 3.5. Standard simplex GPU implementation model 27

4. Measurements and analysis 29
- 4.1. Performance model validation 29
- 4.2. Performance study of our Simplex implementations 29
- 4.3. Performance study of our parallel B&B implementation 32

5. Conclusion and perspectives **37**

II. Fluid simulation in the hearing organ **43**

1. Introduction **47**
1.1. State of the art 47
1.2. Cochlea : the hearing organ 48
1.2.1. Anatomy 48
1.2.2. The hearing mechanism 48
1.3. Computational fluid simulations 50
1.3.1. Palabos, an open source highly parallel solver 50
1.3.2. Palabos elastic shell model 50

2. Simulation of the vestibular duct **53**
2.1. Impact of the duct geometry 53
2.1.1. Rectangular or cylindrical ducts 54
2.1.2. Structural characteristics of the vestibular duct 56
2.2. Emulating the human cochlea conditions 59
2.2.1. Attenuation and absorption of waves 60
2.2.2. Measuring Reissner membrane deformation 61

3. Simulation of the cochlear duct and the organ of Corti **63**
3.1. Coupling both simulations 64
3.2. Velocity of the fluid in the scala media 65

4. Discussion and Conclusion **69**
4.1. Inner ear mechanism 69
4.2. Simulations limitations and challenges 70
4.3. Conclusion 71

III. Toward a high performance framework for statistical inference in evolutionary biology **77**

1. Introduction **81**
1.1. State-of-the-art 81
1.2. Overview 84
1.2.1. Models 84
1.2.2. Methods 84
1.2.3. Large scale analyses 85
1.2.4. Overall framework 85
1.3. Illustrative applications 85
1.3.1. Multivariate normal based models 86
1.3.2. PyRate 86

1.3.3. Tree-based models of molecular evolution 86
1.3.4. Detecting positive selection on protein coding genes 90
1.3.5. Assessing coevolution between pairs of sequence positions 92
1.3.6. Inferring phylogenetic trees 93

2. Model representation and likelihood computation 97
2.1. Why directed acyclic graphs ? 97
2.2. A short introduction to DAGs 98
2.2.1. Directed task graph 99
2.2.2. Graphical model 100
2.2.3. Phylogenetic tree likelihood 100
2.3. Partial Likelihood Update 104
2.3.1. Improvements over `FastCodeML` 107
2.4. Parallel likelihood evaluations 113
2.4.1. Experiments on the branch-site model 116
2.5. Summary 121

3. Maximum likelihood estimation 123
3.1. Basic concepts of continuous optimization 124
3.1.1. Computing the derivatives 125
3.2. Finite difference on DAG 127
3.3. Perturbations scheduling 130
3.3.1. Experiments on the branch-site model (cont.) 131
3.4. Parallel gradient approximations 136
3.4.1. Scheduling and load balancing 140
3.4.2. Experiments on the branch-site model (cont.) 141
3.5. Summary 148

4. Bayesian inference methods 151
4.1. A short introduction to MCMC methods 152
4.1.1. Definition 152
4.1.2. Practical aspects 153
4.2. A novel parallel Metropolis-Hastings method 159
4.2.1. Motivations 159
4.2.2. `EMPIR`: an efficient multivariate adaptive proposal 160
4.2.3. `P-EMPIR`: Securing an optimal pre-fetching with `EMPIR` 162
4.3. Implementation of `EMPIR` and `P-EMPIR` in `HOGAN` 164
4.3.1. Optimising the efficiency of `EMPIR` 165
4.3.2. Managing unbalanced likelihood evaluations in `P-EMPIR` 171
4.3.3. Dealing with multimodal posterior distributions 175
4.4. Assessing `EMPIR` and `P-EMPIR` performance 176
4.4.1. General experimental settings 177
4.4.2. Validation on multivariate normal based models 178
4.4.3. Performance gain on `PyRate` model 180

4.4.4. Performance gain on codon-substitution models . . . 185
4.4.5. Convergence of phylogeny inference . . . 187
4.5. A step toward an automated block creation . . . 191
4.5.1. Formalization as an optimization problem . . . 191
4.5.2. A hierarchical clustering based heuristic . . . 194
4.5.3. Adaptive parameter blocks creation . . . 197
4.6. Summary . . . 201

5. Large scale analyses **203**
5.1. Dynamic load balancing for HPC . . . 204
5.1.1. A receiver initiated diffusion algorithm . . . 204
5.1.2. Implementation in `HOGAN` . . . 207
5.2. Building a database of coevolution . . . 211
5.2.1. Pipeline . . . 211
5.2.2. Current state of `Coev` database . . . 213
5.3. Summary . . . 215

6. Conclusion **217**
6.1. `HOGAN` in a nutshell . . . 217
6.2. Future perspectives . . . 222

Appendices **239**

A. Algorithms **241**
A.1. Dynamic load balancing . . . 241

B. Theorems **243**
B.1. Proof of strict triangle inequality for the scheduling of gradient approximations . . . 243

C. Detailed MCMC experiments **245**
C.1. Codon-substitution models . . . 245
C.1.1. Settings . . . 245
C.1.2. Measures . . . 245
C.2. Convergence on phylogeny inference . . . 246
C.2.1. Settings . . . 246
C.2.2. Measures . . . 246

Part I.

Mixed Integer Programming on GPUs: A Case Study

Preamble

Linear programming rose from being a qualitative tool in the analysis of economic phenomena to one of nowadays most widely used optimisation method. This was made possible by the key contribution of George Dantzig in 1947: the Simplex algorithm [5]. This algorithm, one of the top ten algorithms of the twentieth century [12], remains a primary computational tool in linear and mixed integer programming.

Computational challenges in mixed integer programming evolved from modelling the most cost efficient diet for an active man [41] to finding optimal re-location of factories or optimal schedule for airline company [30]. Originally, efforts required to solve problems such as the diet problem were estimated in man-days [5], nowadays far more complex models only take few seconds to solve using commercial software running on a desktop computer. While a significant progress has been made, large instance of mixed integer programming models may still take several hours, or days to solve [38].

In addition to algorithmic improvements, mixed integer programming solver largely benefited from the considerable increase in processor computing power. However in the last 10 years, processors single-threaded performance started to stagnate creating thus a shift of trend toward a more frequent use of parallel computing [6]. Graphical processing units (GPU), being made of a large amount of processing units, became then a promising perspective for the development of massively parallel applications [37].

In this thesis part, we explore the potential increase in performance that GPUs could bring to mixed integer programming and more particularly to the Simplex algorithm. This study aims at answering two questions:

- How could the Simplex algorithm be implemented on GPUs ?
- Under which conditions could such implementation become competitive with state-of-the-art sequential open-source implementations ?

This part is organized as follow. The first chapter starts with a brief state of the art and then introduces two key method used to solve mixed integer programming models: the Simplex and branch-and-bound algorithms. The second chapter presents the required steps to implement these methods on GPUs. The third chapter proposes a performance model of our standard Simplex implementation. This model is then validated on synthetic datasets in the next chapter followed by an analysis of the performance of our implementations on a well-known benchmark. We conclude in the final chapter by discussing the remaining potential improvements and limitations of the presented implementations.

1. Introduction

1.1. State of the art

The simplex method [11] is a well-known optimization algorithm for solving linear programming (LP) models in the field of operations research. It is part of software often employed by businesses for finding solutions to problems such as airline scheduling problems. The original standard simplex method was proposed by Dantzig in 1947. A more efficient method, named the revised simplex, was later developed (see e.g. [39]). Solving LP models is considered as *easy* using these methods and is categorized in the P class complexity.

However, if some parameters of a LP model would be constrained to be integer values, the effort required to solve such constrained problem would drastically increase. Indeed, these constrained LP models, known as integer linear programming (ILP) models or mixed integer integer programming (MIP) models are categorized as NP-hard. The Branch-and-Bound (B&B) method is a well-known optimization algorithm for solving this class of problems in the field of operations research. It is part of software often employed by businesses for finding solutions to problems such as airline scheduling problems. It operates according to a divide-and-conquer principle by building a tree-like structure with nodes that represent linear programming (LP) problems. More precisely, this tree-based exploration method subdivides the feasible region of the relaxed LP model into successively smaller subsets to obtain bounds on the objective value. Each corresponding submodel can be solved with an LP solver, and the bounds computed determine whether further branching is required. Nowadays its sequential implementation can be found in almost all commercial LP solvers.

However the always increasing complexity and size of LP problems from the industry, drives the demand for more computational power. Indeed, current implementations of the revised simplex, and beyond of B&B, strive to produce the expected results, if any. In this context, parallelization is the natural idea to investigate [29]. Already in 1996, Thomadakis and Liu [44] implemented the standard method on a massive parallel computer and obtained an increase in performances when solving dense or large problems. Parallel B&B tree exploration has been implemented since many years on conventional parallel computers [23], [26]. It remains an active research field [9], [36].

A few years back, in order to meet the demand for processing power, graphics card vendors made their graphical processing units (GPU) available for general-purpose computing. Since then GPUs have gained a lot of popularity as they offer an opportunity to accelerate algorithms having an architecture well-adapted to the GPU model. The simplex method falls into this category. Hence, a natural parallelization of the B&B method is to let the CPU manage the B&B tree and despatch the relaxed submodels to

LP solvers on a GPU. Indeed, GPUs exhibit a massively parallel architecture optimized for matrix processing. To our knowledge, there are only few simplex implementations on GPU. Bieling et al. [4] presented encouraging results while solving small to mid-sized LP problems with the revised simplex (see also [40] and [25]). Following the steps of this first work, an implementation of the revised simplex [16] showed interesting results on dense and square matrices. Finally, an implementation of the interior point method [31] outperformed its CPU equivalent on mid-sized problems. Otherwise, several studies investigate B&B algorithms on GPUs [8], [7], [10], but to our knowledge none in combination with the revised simplex.

For that matter, we present a hybrid CPU-GPU implementation of the B&B algorithm. The B&B tree is managed by the CPU while implementations of the standard and revised simplex methods rely on the CUDA technology of NVIDIA. Let us mention that there are many technicalities involved in CUDA programming, in particular regarding the management of tasks and memories on the GPU. Thus, fine tuning is indispensable to avoid a breakdown on performance.

The following chapters are organized as follows. First, we continue this introduction with a description of the standard and revised simplex methods and introduce the heuristics used in our implementations. This is followed by a presentation of the B&B algorithm and technical considerations about CUDA. The following chapter focus on the GPU implementations of the simplex method as well as the B&B algorithm. In the third chapter, the performance models of the standard simplex implementation is derived and described. This performance model is validated in the forth chapter and a performance comparison between our implementations is made on real-life problems and an analysis is given. Finally, we summarize the results obtained and consider new perspectives.

1.2. Simplex algorithm

1.2.1. Linear programming model

An LP model in its canonical form can be expressed as the following optimization problem:

$$\begin{array}{ll} \text{maximize} & z = \mathbf{c^T x} \\ \text{subject to} & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array} \tag{1.1}$$

where $\mathbf{x} = (x_j), \mathbf{c} = (c_j) \in \mathbb{R}^n, \mathbf{b} = (b_i) \in \mathbb{R}^m$, and $\mathbf{A} = (a_{ij})$ is the $m \times n$ constraints matrix. The objective function $z = \mathbf{c^T x}$ is the inner product of the cost vector $\mathbf{c}$ and the unknown variables $\mathbf{x}$. An element $\mathbf{x}$ is called a solution which is said to be feasible if it satisfies the m linear constraints imposed by $\mathbf{A}$ and the bound $\mathbf{b}$. The feasible region $\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{Ax} \leq \mathbf{b}, \, \mathbf{x} \geq \mathbf{0}\}$ is a convex polytope. An optimal solution to the LP problem will therefore reside on a vertex of this polytope.

1.2.2. Standard simplex algorithm

The simplex method [11] is an algorithm for solving LP models. It proceeds by iteratively visiting vertices on the boundary of the feasible region. This amounts to performing algebraic manipulations on the system of linear equations.

We begin by reformulating the model. So-called *slack variables* x_{n+i} are added to the canonical form in order to replace inequalities by equalities in equation (1.1):

$$x_{n+i} \quad = \quad b_i - \sum_{j=1}^{n} a_{ij}x_j \qquad (i = 1, 2, ..., m) \tag{1.2}$$

The resulting problem is called the *augmented form* in which the variables are divided into two disjoint index sets, $\mathcal{B}$ and $\mathcal{N}$, which correspond to the basic and nonbasic variables. Basic variables, which form the basis of the problem, are on the left-hand side of equation (1.2), while nonbasic variables, which form the core of the equations, appear on the right-hand side. We can thus consider the following LP form:

$$\begin{array}{ll} \text{maximize} & z = \mathbf{c^T x} \\ \text{subject to} & \mathbf{Ax = b} \\ & \mathbf{x \geq 0} \end{array} \tag{1.3}$$

where $\mathbf{x} \in \mathbb{R}^{n+m}$ and $\mathbf{A}$ is now the $m \times (n+m)$ matrix obtained by concatenating the constraints matrix with the $m \times m$ identity matrix $\mathbf{I_m}$. The cost vector has been padded with zeros so that $\mathbf{c} \in \mathbb{R}^{n+m}$.

The basic and nonbasic variables can be separated given the expression

$$\mathbf{A}_{\mathcal{N}}\mathbf{x}_{\mathcal{N}} + \mathbf{x}_{\mathcal{B}} = \mathbf{b}.$$

Similarly, $z = \mathbf{c^T x} = \mathbf{c}_{\mathcal{N}}^{\mathbf{T}}\mathbf{x}_{\mathcal{N}} + \mathbf{c}_{\mathcal{B}}^{\mathbf{T}}\mathbf{x}_{\mathcal{B}}$ with $\mathbf{c}_{\mathcal{B}} = \mathbf{0}$. By definition, a basic solution is obtained by assigning null values to all nonbasic variables ($\mathbf{x}_{\mathcal{N}} = \mathbf{0}$). Hence, $\mathbf{x} = \mathbf{x}_{\mathcal{B}} = \mathbf{b}$ is a basic solution.

The simplex algorithm searches for the optimal solution through an iterative process. For the sake of simplicity, we will assume here that $\mathbf{b} > \mathbf{0}$, that is, the origin belongs to the feasible region. Otherwise a preliminary treatment is required to generate a feasible initial solution (see chapters 1.2.4). A typical iteration then consists of three operations (summarized in Algorithm 1).

1. **Choosing the entering variable.** The entering variable is a nonbasic variable whose increase will lead to an increase in the value of the objective function z. This variable must be selected with care so as to yield a substantial leap towards the optimal solution. The standard way of making this choice is to select the variable x_e with the largest positive coefficient $c_e = \max\{c_j > 0 \,|\, j \in \mathcal{N}\}$ in the objective function z. However, other strategies, such as choosing the positive coefficient with the smallest index, prove to be useful.

2. **Choosing the leaving variable.** This variable is the basic variable which first violates its constraint as the entering variable x_e increases. The choice of the

leaving variable must guarantee that the solution remains feasible. More precisely, setting all nonbasic variables except x_e to zero, x_e is bounded by

$$t = \min \left\{ \frac{b_i}{a_{ie}} \,\middle|\, a_{ie} > 0,\, i = 1, \dots, m \right\}$$

If $t = +\infty$, the LP problem is unbounded.
Otherwise, $t = \dfrac{b_\ell}{a_{\ell e}}$ for some $\ell \in \mathcal{B}$, and x_ℓ is the leaving variable; the element $a_{\ell e}$ is called the pivot.

3. **Pivoting.** Once both variables are defined, the pivoting operation switches these variables from one set to the other: the entering variable enters the basis $\mathcal{B}$, taking the place of the leaving variable which now belongs to $\mathcal{N}$, namely, $\mathcal{B} \leftarrow (\mathcal{B} \setminus \{\ell\}) \cup \{e\}$ and $\mathcal{N} \leftarrow (\mathcal{N} \setminus \{e\}) \cup \{\ell\}$. Correspondingly, the columns with index ℓ and e are exchanged between $\mathbf{I_m}$ and $\mathbf{A}_\mathcal{N}$, and similarly for $\mathbf{c}_\mathcal{B} = \mathbf{0}$ and $\mathbf{c}_\mathcal{N}$. We then update the constraints matrix $\mathbf{A}$, the bound $\mathbf{b}$ and the cost $\mathbf{c}$ using Gaussian elimination. More precisely, denoting $\tilde{\mathbf{I}}_\mathbf{m}$, $\tilde{\mathbf{A}}_\mathcal{N}$, $\tilde{\mathbf{c}}_\mathcal{B}$, and $\tilde{\mathbf{c}}_\mathcal{N}$ as the resulting elements after the exchange, Gaussian elimination then transforms the tableau

$\tilde{\mathbf{A}}_\mathcal{N}$	$\tilde{\mathbf{I}}_\mathbf{m}$	$\mathbf{b}$
$\tilde{\mathbf{c}}_\mathcal{N}^\mathbf{T}$	$\tilde{\mathbf{c}}_\mathcal{B}^\mathbf{T}$	z

into a tableau with updated values for $\mathbf{A}_\mathcal{N}$, $\mathbf{c}_\mathcal{N}$, and $\mathbf{b}$

$\mathbf{A}_\mathcal{N}$	$\mathbf{I_m}$	$\mathbf{b}$
$\mathbf{c}_\mathcal{N}^\mathbf{T}$	$\mathbf{c}_\mathcal{B}^\mathbf{T}$	$z - tc_e$

The latter is obtained by first dividing the ℓ-th row by $a_{\ell e}$; the resulting row multiplied by a_{ie} $(i \neq \ell)$ is then subtracted to the ith row; the same operation is performed using c_e and the last row. Hence, $\mathbf{c}_\mathcal{B} = \mathbf{0}$. These operations amount to jumping from the current vertex to an adjacent vertex with objective value $z = tc_e$.

The algorithm ends when no more entering variable can be found, that is, when $\mathbf{c}_\mathcal{N} \leq \mathbf{0}$.

1.2.3. Revised simplex method

The operation that takes the most time in the standard method is the pivoting operation, and more specifically, the update of the constraints matrix $\mathbf{A}$. The revised method tries to avoid this costly operation by updating only a smaller part of this matrix.

The revised simplex method uses the same operations as in the standard method to choose the entering and leaving variables. However, since the constraints matrix $\mathbf{A}$ need not be fully updated, some additional reformulation is required.

At any stage of the algorithm, basic and nonbasic variables can be separated according to $\mathbf{Ax} = \mathbf{A}_\mathcal{N}\mathbf{x}_\mathcal{N} + \mathbf{A}_\mathcal{B}\mathbf{x}_\mathcal{B} = \mathbf{b}$. We can then transform the system $\mathbf{Ax} = \mathbf{b}$ into $\mathbf{A}_\mathcal{B}\mathbf{x}_\mathcal{B} = \mathbf{b} - \mathbf{A}_\mathcal{N}\mathbf{x}_\mathcal{N}$. Let us denote $\mathbf{B} = \mathbf{A}_\mathcal{B}, \mathbf{N} = \mathbf{A}_\mathcal{N}$. Since $\mathbf{B}$ is invertible, we can write

$$\begin{aligned} \mathbf{x}_\mathcal{B} &= \mathbf{B^{-1}b} - \mathbf{B^{-1}N}\mathbf{x}_\mathcal{N} \\ z &= \mathbf{c}_\mathcal{B}^\mathbf{T}\mathbf{B^{-1}b} + (\mathbf{c}_\mathcal{N}^\mathbf{T} - \mathbf{c}_\mathcal{B}^\mathbf{T}\mathbf{B^{-1}N})\mathbf{x}_\mathcal{N} \end{aligned}$$

Algorithm 1 Standard simplex algorithm

```
//1. Find entering variable
```
if $\mathbf{c}_{\mathcal{N}} \leq \mathbf{0}$ **then**
  return Optimal
end if
Choose an index $e \in \mathcal{N}$ such that $c_e > 0$
```
//2. Find leaving variable
```
if $(\mathbf{A}_{\mathcal{N}})_e \leq \mathbf{0}$ **then**
  return Unbounded
end if
Let $\ell \in \mathcal{B}$ be the index such that

$$t := \frac{b_\ell}{a_{\ell e}} = \min \left\{ \frac{b_i}{a_{ie}} \,\middle|\, a_{ie} > 0,\, i = 1, \ldots, m \right\}$$

```
//3. Pivoting - update
```
$\mathcal{B} := (\mathcal{B} \setminus \{\ell\}) \cup \{e\}$, $\mathcal{N} := (\mathcal{N} \setminus \{e\}) \cup \{\ell\}$
Compute $z_{best} := z_{best} + tc_e$
Exchange $(\mathbf{I_m})_\ell$ and $(\mathbf{A}_{\mathcal{N}})_e$, c_ℓ and c_e
Update $\mathbf{A}_{\mathcal{N}}$, $\mathbf{c}_{\mathcal{N}}$ and $\mathbf{b}$
Go to 1.

The vector $\mathbf{c}_{\mathcal{N}}^{\mathbf{T}} - \mathbf{c}_{\mathcal{B}}^{\mathbf{T}}\mathbf{B}^{-1}\mathbf{N}$ is called the reduced cost vector.

The choice of the leaving variable can be rewritten. Setting all nonbasic variables except the entering variable x_e to zero, x_e is then bounded by

$$t = \min \left\{ \frac{(\mathbf{B}^{-1}\mathbf{b})_i}{(\mathbf{B}^{-1}\mathbf{N})_{ie}} \,\middle|\, (\mathbf{B}^{-1}\mathbf{N})_{ie} > 0,\, i = 1, \ldots, m \right\}$$

If $t = +\infty$, the LP problem is unbounded. Otherwise, $t = \frac{(\mathbf{B}^{-1}\mathbf{b})_\ell}{(\mathbf{B}^{-1}\mathbf{N})_{\ell e}}$ for some $\ell \in \mathcal{B}$, and x_ℓ is the leaving variable.

Recall that the pivoting operation begins by exchanging columns with index ℓ and e between $\mathbf{B}$ and $\mathbf{N}$ and similarly for $\mathbf{c}_{\mathcal{B}}$ and $\mathbf{c}_{\mathcal{N}}$.

To emphasize the difference between standard and revised simplex, we first express the updating for the standard simplex. This amounts to

$$\begin{aligned} [\mathbf{B} \quad \mathbf{N} \quad \mathbf{b}] &\leftarrow [\mathbf{I_m} \quad \mathbf{B}^{-1}\mathbf{N} \quad \mathbf{B}^{-1}\mathbf{b}] \\ [\mathbf{c}_{\mathcal{B}}^{\mathbf{T}} \quad \mathbf{c}_{\mathcal{N}}^{\mathbf{T}}] &\leftarrow [\mathbf{0} \quad \mathbf{c}_{\mathcal{N}}^{\mathbf{T}} - \mathbf{c}_{\mathcal{B}}^{\mathbf{T}}\mathbf{B}^{-1}\mathbf{N}] \end{aligned}$$

which moves the current vertex to an adjacent vertex $\mathbf{x} = \mathbf{B}^{-1}\mathbf{b}$ with objective value $z = \mathbf{c}_{\mathcal{B}}^{\mathbf{T}}\mathbf{B}^{-1}\mathbf{b}$ (the end condition remains $\mathbf{c}_{\mathcal{N}} \leq \mathbf{0}$). Many computations performed in this update phase can in fact be avoided.

The main point of the revised simplex method is that the only values which really need to be computed at each step are $\mathbf{B}^{-1}$, $\mathbf{B}^{-1}\mathbf{b}$,$\mathbf{B}^{-1}\mathbf{N}_e$, $\mathbf{c}_{\mathcal{B}}^{\mathbf{T}}\mathbf{B}^{-1}$, and $\mathbf{c}_{\mathcal{B}}^{\mathbf{T}}\mathbf{B}^{-1}\mathbf{b}$. However, matrix inversion is a time-consuming operation (of cubic order for Gaussian elimination). Fortunately, there are efficient ways of computing an update for $\mathbf{B}^{-1}$. One way is to take

advantage of the sparsity of the LP problem by using the so-called LU decomposition[1] for sparse matrices [3]. This decomposition may be updated at a small cost at each iteration [42]. Another way is to use the *product form of the inverse* [14], which we describe hereafter.

Let $\mathbf{b}_1, \dots, \mathbf{b}_m$ be the columns of $\mathbf{B}$, $\mathbf{v} \in \mathbb{R}^m$, and $\mathbf{a} = \mathbf{B}\mathbf{v} = \sum_{i=1}^m v_i \mathbf{b}_i$. Denote $\mathbf{B_a} = (\mathbf{b}_1, \dots, \mathbf{b}_{p-1}, \mathbf{a}, \mathbf{b}_{p+1}, \dots, \mathbf{b}_m)$ for a given $1 \leq p \leq m$ such that $v_p \neq 0$. We want to compute $\mathbf{B_a}^{-1}$. We first write

$$\mathbf{b}_p = \frac{1}{v_p} + \mathbf{a} \sum_{i \neq p} \frac{-v_i}{v_p} \mathbf{b}_i$$

Define

$$\boldsymbol{\eta} = \left(\frac{-v_1}{v_p}, \dots, \frac{-v_{p-1}}{v_p}, \frac{1}{v_p}, \frac{-v_{p+1}}{v_p}, \dots, \frac{-v_m}{v_p} \right)^T$$

and

$$\mathbf{E} = (\boldsymbol{\varepsilon}_1, \dots, \boldsymbol{\varepsilon}_{p-1}, \boldsymbol{\eta}, \boldsymbol{\varepsilon}_{p+1}, \dots, \boldsymbol{\varepsilon}_m)$$

where $\boldsymbol{\varepsilon}_j = (0, \dots, 0, 1, 0, \dots, 0)^T$ is the jth element of the canonical basis of $\mathbb{R}^m$. Then $\mathbf{B_a}\mathbf{E} = \mathbf{B}$, so $\mathbf{B_a}^{-1} = \mathbf{E}\,\mathbf{B}^{-1}$.

We apply these preliminary considerations to the simplex algorithm with $\mathbf{a} = \mathbf{N}_e$, $\mathbf{v} = (\mathbf{B^{-1}N})_e$ (recall that x_e is the entering variable). If initially $\mathbf{B}$ is the identity matrix $\mathbf{I_m}$, at the kth iteration of the algorithm, the inverse matrix is given by $\mathbf{B}^{-1} = \mathbf{E}_k \mathbf{E}_{k-1} \cdots \mathbf{E}_2 \mathbf{E}_1$, where $\mathbf{E}_i$ is the matrix constructed at the ith iteration.

1.2.4. Heuristics and improvements

Specific heuristics or methods are needed to improve the performance and stability of the simplex algorithm. We will explain below how we find an initial feasible solution and how we choose the entering and leaving variables.

Finding initial feasible solutions. Our implementations use the *two-phase simplex* [11]. The first phase aims at finding a feasible solution required by the second phase to solve the original problem. If the origin $\mathbf{x} = \mathbf{0}$ is not a feasible solution, we proceed to find such a solution by solving a so-called *auxiliary problem* with the simplex algorithm. This can be achieved by adding a nonnegative artificial variable to each constraint equation corresponding to a basic variable which violates its nonnegativity condition, before minimizing the sum of these artificial variables. The algorithm will thus try to drive all artificial variables towards zero. If it succeeds, then a basic feasible solution is available as an initial solution for the second phase in which it attempts to find an optimal solution to the original problem. Otherwise, the problem is infeasible.

To avoid having to introduce these additional auxiliary variables, we use an alternate version of this procedure. For basic variables with index in $\mathcal{I} = \{j \in \mathcal{B} \mid x_j < 0\}$, we

[1] The LU decomposition is a linear algebra decomposition which allows to write a matrix as a product of a lower and an upper triangular matrix

temporarily relax the nonnegativity condition in order to have a feasible problem and then apply the simplex algorithm to minimize $w = -\sum_{j \in \mathcal{I}} x_j$. However, we update $\mathcal{I}$ at each iteration and modify accordingly the objective function, whose role is to drive these infeasible basic variables towards their original bound. If at some stage $\mathcal{I} = \emptyset$, we end up with an initial feasible solution. Otherwise, the algorithm terminates with $\mathcal{I} \neq \emptyset$ and $w > 0$, which indicates that the original problem is infeasible. The alteration introduced above involves more computations during the first phase, but it offers the advantage of preserving the problem size and making good use of the GPU processing power.

Choice of the entering variable. The number of iterations required to solve a problem depends on the method used to select the entering variable. The one described in chapter 1.2.2 chooses the most promising variable x_e in terms of cost. While being inexpensive to compute, this method can lead to an important number of iterations before the best solution is found.

There exist various heuristics to select this variable x_e. One of the most commonly used is the *steepest-edge* method [24]. To improve the speed at which the best solution is found, this method takes into account the coefficients of $\mathbf{B}^{-1}\mathbf{N}$. This can be explained from the geometrical point of view. The constraints $\mathbf{Ax} = \mathbf{b}$ form the hull of a convex polytope. The simplex algorithm moves from one vertex (i.e., a solution) to another while trying to improve the objective function. The steepest-edge method searches for the edge along which the rate of improvement of the objective function is the best. The entering variable x_e is then determined by

$$e = \arg\max \left\{ \frac{c_j}{\sqrt{\gamma_j}} \,\middle|\, c_j > 0, j \in \mathcal{N} \right\}$$

with $\gamma_j = \|\mathbf{B}^{-1}\mathbf{N}_j\|^2$.

This method is quite costly to compute but it reduces significantly the number of iterations required to solve a problem. This heuristic can be directly applied to the standard simplex since the full tableau is updated at each iteration. However, it defeats the purpose of the revised simplex algorithm since the aim is precisely to avoid updating the whole constraints matrix at each iteration. Taking this into account, the steepest-edge coefficients $\boldsymbol{\gamma}$ are updated based only on their current value. For the sake of clarity, the hat notation is used to differentiate the next iteration value of a variable from its current value: for example if $\boldsymbol{\gamma}$ denotes the steepest-edge coefficients at the current iteration, then $\hat{\boldsymbol{\gamma}}$ are the updated coefficients.

Given the column of the entering variable $\mathbf{d} = (\mathbf{B^{-1}N})_e$, we may process afresh the steepest-edge coefficient of the entering variable as $\gamma_e = \|\mathbf{d}\|^2$. The updated steepest-edge coefficients are then given by

$$\hat{\gamma_j} = \max\left\{\gamma_j - 2\hat{\alpha_j}\beta_j + \gamma_e \hat{\alpha_j}^2, 1 + \hat{\alpha_j}^2\right\} \qquad \text{for } j \neq e$$

$$\hat{\gamma_e} = \gamma_e / d_e^2$$

with $\hat{\boldsymbol{\alpha}} = \mathbf{N^T}((\hat{\mathbf{B}}^{\mathbf{-1}})^{\mathbf{T}})_\ell$ and $\boldsymbol{\beta} = \mathbf{N^T}(\mathbf{B^{-1}})^{\mathbf{T}}\mathbf{d}$

Choice of the leaving variable. The stability and robustness of the algorithm depend considerably on the choice of the leaving variable. With respect to this, the *expand* method [21] proves to be very useful in the sense that it helps to avoid cycles and reduces the risks of encountering numerical instabilities. This method consists of two steps of complexity $\mathcal{O}(m)$. In the first step, a small perturbation is applied to the bounds of the variables to prevent stalling of the objective value, thus avoiding cycles. These perturbed bounds are then used to determine the greatest gain on the entering variable imposed by the most constraining basic variable. The second phase uses the original bounds to define the basic variable offering the gain closest to the one of the first phase. This variable will then be selected for leaving the basis.

1.3. Branch-and-bound algorithm

1.3.1. Integer linear programming

In some problems, variables are integer-valued. For example, in a vehicle routing problem, one cannot assign one third of a vehicule to a specific route. ILP problems restrict LP problems by imposing an integrality condition on the variables. While this change may seem to have little impact on the model, the aftermaths on the resolution method are quite important.

From a geometrical perspective, the simplex algorithm is a method for finding an optimum in a convex polytope defined by an LP problem. However, the additional integrality condition results in the loss of this convexity property. The resolution method must then be altered in order to find a solution to the ILP problem. The idea is to explore the solution space by a divide-and-conquer approach in which the simplex algorithm is used to find a local optimum in subspaces.

1.3.2. Branch-and-bound tree

The solution space is explored by the B&B algorithm [1]. The strategy used is conceptually close to a tree traversal.

At first, the ILP problem is considered as an LP problem by relaxing the integrality condition before applying an LP solver to it. This initial relaxed problem represents the root of the tree about to be built. From the obtained solution $\boldsymbol{\xi}$, if ξ_j is not integral for some j, then x_j can be chosen as a branching variable. The current problem will then be divided, if possible, into two subproblems: one having $x_j \leq \lfloor \xi_j \rfloor$ and the other $x_j \geq \lceil \xi_j \rceil$. Each of these LP subproblems represents a child node waiting to be solved.

This is repeated for each subproblem in such a way that all variables are led towards integral values. At some point a subproblem will either be infeasible or lead to a feasible ILP solution (leaf nodes). The algorithm ends when the tree has been fully visited, returning the best feasible ILP solution.

During the exploration, lower and upper bounds on the objective value are computed. The upper bound is represented by the best objective value encountered in a child node. This nonadmissible solution hints towards what the objective value of the ILP problem

could be. The lower bound is the best ILP solution yet found, in other words the objective value of the most promising leaf node. The bounds may be used to prune subtrees when further branching cannot improve the best current solution.

In the B&B algorithm, the main two operations that impact the convergence of the bounds towards the optimal solution are the branching strategy and the node selection strategy.

Example

Figure 1.1 illustrates the use of the B&B algorithm for solving an ILP. The ILP problem is the following:

$$\begin{aligned} \text{Maximize} \quad & z = x_1 + 4x_2 \\ \text{Subject to} \quad & 5x_1 + 8x_2 \leq 40 \\ & -2x_1 + 3x_2 \leq 9 \\ & x_1, x_2 \geq 0 \quad \text{integer-valued} \end{aligned}$$

The nodes are solved with the simplex method in the order written on the figure. At the 3rd step of the B&B, the first feasible solution is encountered (light grey leaf). The optimal solution is encountered at the 7th step (dark grey leaf). However, this is assessed only after solving the last two nodes (8th and 9th) whose objective value z is inferior to the best one yet found.

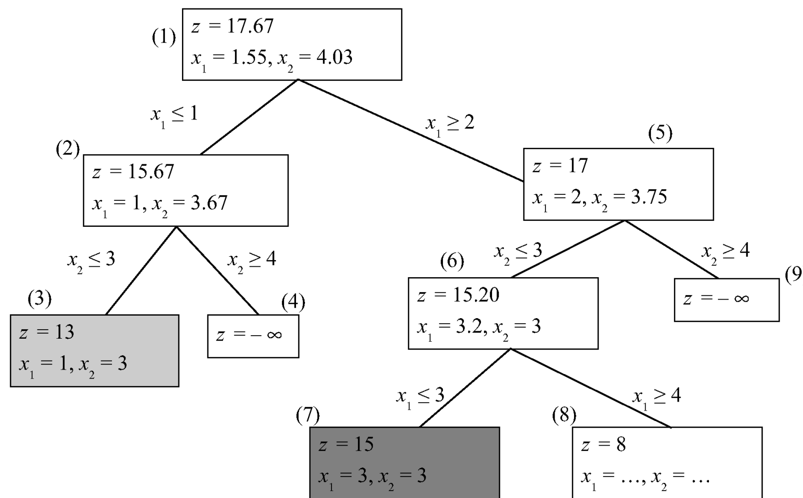


Figure 1.1.: Solving an ILP problem using a branch-and-bound algorithm.

1.3.3. Branching strategy

The branching strategy defines the method used to select the variable on which branching will occur. The objective value of the child node depends greatly on the choice of this variable. Branching on a variable may lead to a drop on the upper bound and thus speed up the exploration, while branching on other variables could leave this bound unchanged.

Several branching strategies exist [32]. Let us briefly comment on some of them in terms of improving the objective function and processing cost.

- *Smallest-index strategy* is a greedy approach that always selects the variable with the smallest index as branching variable. This method is simple, has a cheap processing cost, but does not try to select the best variable.
- *Strong branching strategy* is an exhaustive strategy. The branching variable selected is the one among all the potential variables that leads to the best solution. This means that for each potential branching variable, its potential child nodes must be solved. This method is easy to implement, but its computational cost makes it inefficient.
- *Reliability branching strategy* is a strategy which maintains a pseudocost [20] for each potential branching variable. The pseudocost of a variable is based on the result obtained when branching on it at previous steps. Since at the start, pseudocosts are unreliable, a limited strong branching approach is used until pseudocosts are deemed reliable. This method is more complex than the two others and requires fine tuning. It offers, however, the best trade-off between improvement and computional cost.

1.3.4. Node selection strategy

The node selection strategy defines the methodology used to explore the tree. While the usual depth-first search and breadth-first search are considered and used, some remarks about the tree exploration must be made. First let us mention a few facts:

1. Solutions obtained from child nodes cannot be better than the one of their parent node.
2. An LP solver is able to quickly find a solution if the subproblem (child node) is only slightly different from its parent.

Given the above statements, it is of interest to quickly find a feasible solution. Indeed, this allows the pruning of all pending nodes which do not improve the solution found. However, the quality of the latter solution impacts the amount of nodes pruned. It takes more time to produce a good solution because one must search for the best nodes in the tree. Consequently, a trade-off must be made between the quality and the time required to find a solution.

Two types of strategies [34] can then be considered:

- *Depth-first search* aims at always selecting one of the child nodes until a leaf (infeasible subproblem or feasible solution) is reached. This strategy is characterized by fast solving, and it quickly finds a feasible solution. It mostly improves the lower bound.
- *Best-first search* aims at always selecting one of the most promising nodes in the tree. This strategy is characterized by slower solving but guarantees that the first feasible solution found is the optimal. It mostly improves the upper bound.

A problem occurs with the best-first search strategy: there might be numerous nodes having solutions of the same quality, thus making the choice of a node difficult. To avoid this problem, the *best-estimate search* strategy [22] differentiates the best nodes by estimating their potential cost with the help of a pseudocost (see previous chapter).

The most promising variant is a hybrid strategy in which the base strategy is the best-estimate search with the subtrees of the best-estimated nodes being then subject to a limited depth-first search, more commonly called *plunging*. This method launches a fast search for a feasible solution in the most promising subtrees, thus improving the upper and lower bounds at the same time.

1.3.5. Cutting-plane methods

Cutting-planes [46] (also simply *cuts*) are new constraints whose role is to cut off parts of the search space. They may be generated from the first LP solution (*cut-and-branch*) or periodically during the B&B (*branch-and-cut*). On the one hand cutting-planes may considerably reduce the size of the solution space, but on the other hand they increase the problem size. Moreover, generating cutting-planes is costly since it requires a thorough analysis of the current state of the problem.

Various types or families of cutting-planes exist. Those that are most applied are the *Gomory cutting-planes* and the *complemented mixed integer rounding inequalities* (c-MIR). Other kinds of cutting-planes target specific families of problems, for example, the *0-1 knapsack cutting-planes* or *flow cover cuts* [35].

The cutting-planes generated must be carefully selected in order to avoid a huge increase in the problem size. They are selected according to three criteria: their efficiency, their orthogonality with respect to other cutting-planes, and their parallelism with respect to the objective function. Cutting-planes having the most impact on the problem are then selected, while the others are dropped.

1.4. CUDA considerations

The most expensive operations in the simplex algorithm are linear algebra functions. The CUBLAS library is used for dense matrix-vector multiplications (`cublasDgemv`) and dense vector sums (`cublasDaxpy`). The CUSPARSE library is used for the sparse matrix-vector multiplication (`cusparseDcsrmv`).

However, complex operations are required to implement the simplex and must be coded from scratch. For example, reduction operations (*argmax, argmin*) are fundamental

for selecting variables. In the following chapter, we first quickly describe the CUDA reduction operation, before making some global remarks on kernel optimization.

1.4.1. Parallel reduction

A parallel reduction operation is performed in an efficient manner inside a GPU block as shown in Figure 1.2. Shared memory is used for a fast and reliable way to communicate between threads. However, at the grid level, reduction cannot be easily implemented due to the lack of direct communication between blocks. The usual way of dealing with this type of limitation is to apply the reduction in two separate steps. The first one involves a GPU kernel reducing the data over multiple blocks, the local result of each block being stored on completion. The second step finishes the reduction on a single block or on the CPU.

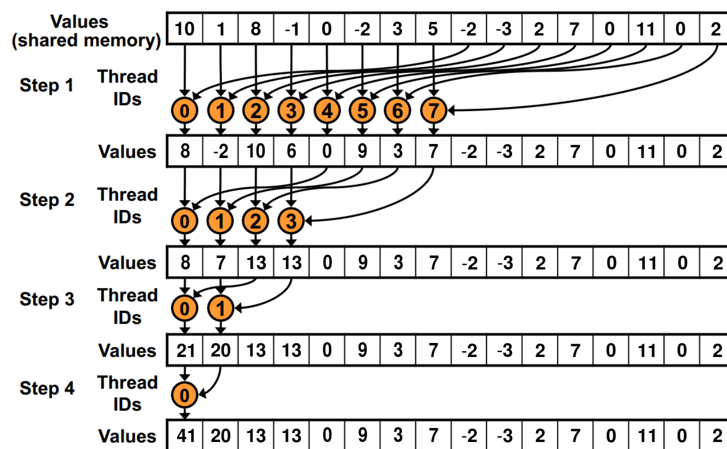


Figure 1.2.: Example of a parallel reduction at block level (courtesy NVIDIA).

An optimized way of doing the reduction can be found in the example provided by NVIDIA. In order to keep code listings compact hereafter, the reduction of values among a block will be referred to as *reduceOperation(value)* (per extension *reduceArgMax(maxVal)*).

1.4.2. Kernel optimization

Optimizing a kernel is a difficult task. The most important point is to determine whether the performances are limited by the bandwidth or by the instruction throughput. Depending on the case and the specificities of the problem, various strategies may be applied. This part requires a good understanding of the underlying architecture and its limitations. The *CUDA Programming Guide* offers some insight into this subject, as do the interesting articles and presentations by Vassily Volkov [45]. The CUDA profiler is the best way to monitor the performances of a kernel. This tool proposes multiple performance markers giving indications about potential bottlenecks.

2. Implementations

In this chapter, the implementations of the simplex algorithm are studied. The emphasis is put on the main algorithms and kernels having a major impact on performance. The *expand* method used for choosing the leaving variable will not be detailed for that reason. Then the B&B implementation is quickly explained focusing on the interaction between the B&B algorithm and the simplex solver.

2.1. Standard simplex

The implementation of the standard simplex algorithm (see chapter 1.2.2) is rather straightforward. The main difference with Algorithm 1 is that the basis matrix is not stored in memory since it is equal to the identity matrix most of the time. Instead a proper data structure keeps track of the basic variables and their values.

Algorithm 2 Standard simplex algorithm

// Find entering variable e
$\gamma_j \leftarrow \|(\mathbf{A}_\mathcal{N})_j\|^2$
$e \leftarrow argmax\ (\mathbf{c}/\sqrt{\gamma})$
if $e < 0$ **then**
  return `optima_found`
end if
// Find leaving variable ℓ
$\ell, t \leftarrow expand((\mathbf{A}_\mathcal{N})_e)$
if $\ell < 0$ **then**
  return `unbounded`
end if
// Pivoting
$\mathbf{d} \leftarrow (\mathbf{A}_\mathcal{N})_e$, $d_e \leftarrow (A_\mathcal{N})_{\ell e} - 1$
$\mathbf{r} \leftarrow \mathbf{N^T}_\ell$
$x_e \leftarrow x_e + t$
$\mathbf{x}_\mathcal{B} \leftarrow \mathbf{x}_\mathcal{B} - t(\mathbf{A}_\mathcal{N})_e$
$\mathbf{A}_\mathcal{N} \leftarrow \mathbf{A}_\mathcal{N} - \mathbf{d^T}\mathbf{r}/(A_\mathcal{N})_{\ell e}$ *// cublasDger*
$\mathbf{c} \leftarrow \mathbf{c} - c_\ell \mathbf{r}/(A_\mathcal{N})_{\ell e}$ *// cublasDaxpy*
$swap(x_e, x_\ell)$

The first step of Algorithm 2 is the selection of the entering variable using the steepest-edge heuristic. This step is discussed thoroughly in the next chapter. Then the leaving

variable index ℓ and the potential gain on the entering variable t are determined using the *expand* method. Finally, the pivoting is done. The nonbasic matrix $\mathbf{A}_{\mathcal{N}}$ and the objective function coefficients $\mathbf{c}$ are updated using the CUBLAS library (respectively, with `cublasDger` and `cublasDaxpy`). This requires the entering variable column $\mathbf{d}$ and the leaving variable row $\mathbf{r}$ to be copied and slightly altered. The new value of the variables is computed and, since we use a special structure instead of the basis matrix, the entering and leaving variables are swapped.

Choice of the entering variable

Let us discuss two different approaches for the selection of the entering variable. The first one relies on the CUBLAS library. The idea is to split this step into several small operations, starting with the computation of the norms one by one with the `cublasDnrm2` function. The score of each variable is then obtained by dividing the cost vector $\mathbf{c}$ by the norms previously computed. The entering variable is finally selected by taking the variable with the biggest steepest-edge coefficient using `cublasDamax`.

While being easy to implement, such an approach would lead to poor performances. The main problem is a misuse of data parallelism. Each column can be processed independently, and thus at the same time. Moreover, slicing this step into small operations requires that each temporary result be stored in global memory. This creates a huge amount of slow data transfers between kernels and global memory.

Listing 2.1: a kernel for the choice of the entering variable

```
extern __shared__ volatile double sData[];
__global__ void
selectInVar(int m, int n, double *c, double *AN, uint pitchAN,
            uint *resIdx, double *resVal) {
  uint i, maxIdx = -1, bid = blockIdx.x;
  double val, locSum, xScore, maxScore = 0.0;
  while (bid < n) { // Processing multiple columns
    i = threadIdx.x;
    locSum = 0.0;
    if (isPotentialEnteringVar(bid)) { // Do the local processing
      while (i < m) { // Each thread processes multiple elements
        val = AN[i+bid*pitchAN];
        locSum += val*val;
        i += blockDim.x;
      }
      // Reduce the value using shared memory
      reduceSum(locSum);
      if (tid == 0){ // Is this the best variable encountered ?
        // on tid=0 locSum equals the steepest edge coeffcient
        xScore = cVal*rsqrt(locSum);
        if (fabs(maxScore) < fabs(xScore)) {
          maxIdx = bid;
          maxScore = xScore;
        }
      }
      __syncthreads();
```

```
27      }
28      bid += gridDim.x;
29    }
30    // Write the result into global memory
31    if (tid == 0) {
32      resIdx[blockIdx.x] = maxIdx;
33      resVal[blockIdx.x] = maxScore;
34    }
35  }
```

To avoid this, the whole step could be implemented as a unique kernel, as shown in the simplified Listing 2.1. Each block evaluates multiple potential entering variables. The threads of a block process part of the norm of a column. Then a reduction (see chapter 1.4.1) is done inside the block to form the full norm. The thread having the final value can then compute the score of the current variable and determine if it is the best one encountered in this block. Once a block has evaluated its variables, the most promising one is stored in global memory. The final reduction step is finally done on the CPU.

The dimension of the blocks and grid have to be chosen wisely as a function of the problem size. The block size is directly correlated to the size of a column while the grid size is a trade-off between giving enough work to the scheduler in order to hide the latency and returning as few potential entering variables as possible for the final reduction step.

CPU-GPU interactions

The bulk of the data representing the problem is stored on the GPU. Only variables required for decision-making operations are updated on the CPU. The communications arising from the aforementioned scheme are illustrated in Figure 2.1. The amount of data exchanged at each iteration is independent of the problem size ensuring that this implementation scales well as the problem size increases.

2.2. Revised simplex

The main steps found in the previous implementation are again present in the revised simplex. The difference comes from the fact that a full update of the matrix $\mathbf{N}$ is not necessary at every iteration. As explained in chapter 1.2.3, the basis matrix $\mathbf{B}$ is updated so that the required values, such as the entering variable column, can be processed from the now *constant* matrix $\mathbf{N}$. Due to these changes, the steepest-edge method is adapted. The resulting implementation requires more operations as described in Algorithm 3.

Let us compare the complexity, in terms of level 2 and level 3 BLAS operations, of both implementations. The standard one has mainly two costly steps: the selection of the entering variable and the update of the matrix $\mathbf{N}$. These are level 2 BLAS operations or the equivalent, and thus the approximate complexity is $\mathcal{O}(2mn)$.

The new implementation proposed has three operations `cublasDgemv` where the matrix $\mathbf{N}$ is involved, and three others with the matrix $\mathbf{B^{-1}}$. The complexities of these oper-

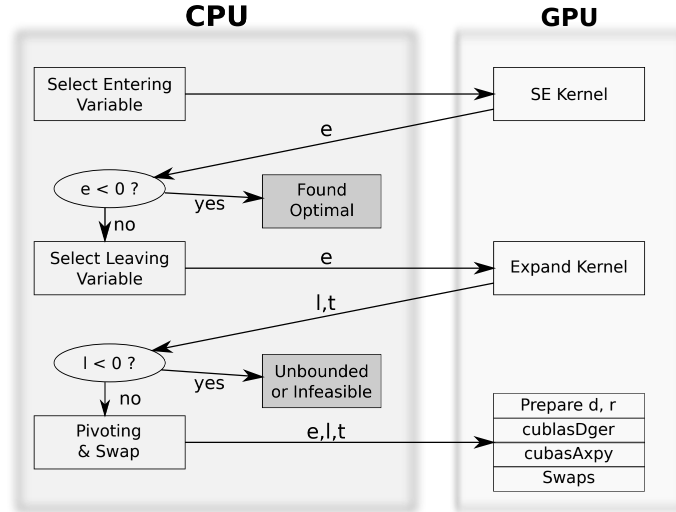


Figure 2.1.: Communications between CPU and GPU.

ations are respectively $\mathcal{O}(mn)$ and $\mathcal{O}(m^2)$. The update of the matrix $\mathbf{B}$ is a level 3 BLAS operation costing $\mathcal{O}(m^3)$. The approximate complexity is then $\mathcal{O}(3mn + 3m^2 + m^3)$.

It appears clearly that, in the current state, the revised simplex implementation is less efficient than the standard one. However this method can be improved and might well be better than the standard one based on two characteristics of LP problems: their high sparsity and the fact that usually $m \ll n$. In the next chapters, we will give details about these improvements.

Choice of the entering variable

Once again, this part of the algorithm can be substantially improved. First, algorithmic optimizations must be considered. Since row $\boldsymbol{\alpha}$ of the leaving variable is processed to update the steepest-edge coefficients, the cost vector $\boldsymbol{\upsilon}$ can be updated directly without using the basis matrix $\mathbf{B}$. This is done as follow (see [43])

$$\upsilon_j = \bar{\upsilon}_j - \bar{\upsilon}_e \alpha_j,\ \forall j \neq e, \qquad \upsilon_e = -\frac{\bar{\upsilon}_e}{\alpha_e}$$

where $\bar{\upsilon}_j$ denotes the value of υ_j at the previous iteration. It is important to note that all these updated values $(\boldsymbol{\upsilon}, \boldsymbol{\gamma})$ must be processed afresh once in a while to reduce numerical errors.

Including this change, the operations required for the selection of the entering variable e are detailed in Algorithm 4. The values related to the entering variable at the previous iteration $\bar{e} = q$ are used. The reduced costs are updated, followed by the steepest-edge coefficients. With these values the entering variable is determined.

Coupling these operations into a single kernel is quite beneficial. This leads $\boldsymbol{\upsilon}$ and $\boldsymbol{\gamma}$ to be loaded only once from global memory. Their updated values are stored while the entering variable e is selected. With these changes, the global complexity of this step

Algorithm 3 Revised simplex algorithm

// *Find entering variable* x_e, $e \in \mathcal{B}$
$\boldsymbol{\tau} \leftarrow (\mathbf{B^{-1}})^{\mathbf{T}} \mathbf{c}_{\mathcal{B}}$ // *cublasDgemv*
$\boldsymbol{\upsilon} \leftarrow \mathbf{c}_{\mathcal{N}} - \mathbf{N^T} \boldsymbol{\tau}$ // *cublasDgemv*
$e \leftarrow argmax\ (\boldsymbol{\upsilon}/\sqrt{\boldsymbol{\gamma}})$
if $e < 0$ **return** `optima_found`
// *Find leaving variable* x_ℓ, $\ell \in \mathcal{N}$
$\mathbf{d} \leftarrow \mathbf{B^{-1}} \mathbf{N}_e$ // *cublasDgemv*
$\ell, t \leftarrow expand(\mathbf{d})$
if $\ell < 0$ **return** `unbounded`
// *Pivoting - basis update*
$\boldsymbol{\kappa} \leftarrow (\mathbf{B^{-1}})^{\mathbf{T}} \mathbf{d}$ // *cublasDgemv*
$\boldsymbol{\beta} \leftarrow \mathbf{N^T} \boldsymbol{\kappa}$ // *cublasDgemv*
$\mathbf{B^{-1}} \leftarrow \mathbf{E}\, \mathbf{B^{-1}}$
$x_e \leftarrow x_e + t$
$\mathbf{x}_{\mathcal{B}} \leftarrow \mathbf{x}_{\mathcal{B}} - t\, \mathbf{d}$
$swap(x_\ell, x_e)$
$\hat{\boldsymbol{\alpha}} \leftarrow \mathbf{N^T} \left((\mathbf{B^{-1}})^{\mathbf{T}} \right)_\ell$ // *cublasDgemv*
$\gamma_e \leftarrow \|\mathbf{d}\|^2$
$\hat{\gamma_j} \leftarrow \max \left\{ \gamma_j - 2\hat{\alpha_j}\beta_j + \gamma_e \hat{\alpha_j}^2, 1 + \hat{\alpha_j}^2 \right\},\ \forall j \neq e, \quad \hat{\gamma_e} \leftarrow \gamma_e / d_e^2$

Algorithm 4 Choice of the entering variable

$q \leftarrow \bar{e}$
$\bar{\upsilon}_q \leftarrow c_q - \mathbf{c}_{\mathcal{B}}^{\mathbf{T}} \bar{\mathbf{d}}$
$\bar{\gamma}_q \leftarrow \|\bar{\mathbf{d}}\|$
// *Update of the reduced costs*
$\upsilon_j \leftarrow \bar{\upsilon}_j - \alpha_j \bar{\upsilon}_q,\ \forall j \neq q$
$\upsilon_q \leftarrow \dfrac{\bar{\upsilon}_q}{\bar{d}_q^2}$
// *Update of the steepest edge coefficients*
$\gamma_j \leftarrow \max \left\{ \bar{\gamma}_j - 2\alpha_j \bar{\beta}_j + \bar{\gamma}_q \alpha_j^2, 1 + \alpha_j^2 \right\},\ \forall j \neq q$
$\gamma_q \leftarrow \bar{\gamma}_q / \bar{d}_q^2$
// *Selection of the entering variable*
$e \leftarrow argmax\ (\boldsymbol{\upsilon}/\sqrt{\boldsymbol{\gamma}})$

is reduced from $\mathcal{O}(mn + m^2 + n)$ to $\mathcal{O}(n)$. Moreover the remaining processing is done optimally by reusing data and by overlapping global memory access and computations.

Basis update

The basis update $\mathbf{B^{-1}} \leftarrow \mathbf{E}\, \mathbf{B^{-1}}$ is a matrix-matrix multiplication. However, due to the special form of the matrix $\mathbf{E}$ (see chapter 1.2.3), the complexity of this operation can be

reduced from $\mathcal{O}(m^3)$ to $\mathcal{O}(m^2)$ [18]. The matrix $\mathbf{E}$ is merely the identity matrix having the ℓth column replaced by the vector $\boldsymbol{\eta}$. The update of the matrix $\mathbf{B^{-1}}$ can be rewritten as

$$\hat{B}_{ij}^{-1} = B_{ij}^{-1}\left(1 - \frac{d_i}{d_\ell}\right), \quad \forall i \neq \ell, \qquad \hat{B}_{\ell j}^{-1} = \frac{B_{\ell j}^{-1}}{d_\ell}$$

As shown in Listing 2.2, each block of the kernel processes a single column while each thread may compute multiple elements of a column. This organization ensures that global memory accesses are coalescent since the matrix $\mathbf{B}$ is stored column-wise.

Listing 2.2: basis update

```
extern __shared__ volatile double sdata[];
__global__ void
updateBasisKernel(int m, uint l, double d_l, double *B,
                  uint pitch_B, double *d) {
  uint bId = blockIdx.x, tId = threadIdx.x;
  uint colStart = bId*pitch_B;
  double Bij, d_i, B2ij;

  // First thread loads Blj so it can be
  // broadcast via shared memory to each thread
  if (tId == 0)
    sdata[0] = B[colStart+l] / d_l;
  __syncthreads();

  // Each thread proccesses multiple elements
  while (tId < m) {
    // Load di and Bij
    d_i = d[tId];
    Bij = B[colStart+tId];
    // Update Bij
    B2ij = sdata[0];
    if (tId != q) {
      B2ij *= -d_i;
      B2ij += Bij
    }
    __syncthreads();
    B[colStart+tId] = B2ij;

    tId += blockDim.x;
  }
}
```

Sparse matrix operations

The complexity of the implementation is now $\mathcal{O}(2mn+3m^2)$ which is close to the one of the standard simplex implementation. The operations where the matrix $\mathbf{N}$ is involved remain the more expensive (considering $m \ll n$). However, this matrix is *constant* in the revised simplex allowing the use of sparse matrix-vector multiplication from the CUSPARSE library. On sparse problems, this leads to important gains in performance.

The sparse storage of the matrix **N** reduces significantly the memory space used by the algorithm. All these improvements come at a small cost: columns are no longer directly available in their dense format and must be decompressed to their dense representation when needed.

The complexity of the revised simplex implementation is thus finally $\mathcal{O}(2\theta + 3m^2)$ where θ is a function of m, n, and the sparsity of the problem. We shall see in chapter 4.2, what kind of performances we may obtain on various problems.

2.3. Branch-and-bound

The B&B algorithm is operated from the CPU. The simplex implementations, also referred to as LP solver, are used to solve the nodes of the B&B tree. Algorithm 5 contains the main operations discussed in chapter 1.3. It starts by solving the root node. The branching strategy is then used to select the next variable to branch on. From thereon, until no node remains to be solved, the node selection strategy is used to select the next one to be processed.

The critical factor for coupling the LP solver and the B&B is the amount of communications exchanged between them. Since CPU-GPU communications are expensive, the informations required to solve a new node must be minimized. Upon solving a new node, the full transfer of the problem must be avoided. This will be the focus of this chapter.

Algorithm 5 Branch-and-Bound

```
Solution ← Solve(InitialProblem)
BranchVar ← SelectNextVariable(Solution)
NodeList ← AddNewNodes(BranchVar)
while NodeList ≠ ∅ do
  Node ← SelectNextNode(NodeList)
  Solution ← Solve(Node)
  if exists(Solution) then
    BranchVar ← SelectNextVariable(Solution)
    NodeList ← AddNewNodes(BranchVar)
  end if
end while
```

Algorithmic choices

The first critical choice is to select an efficient tree traversal strategy that also takes advantage of the locality of the problem. At first glance, the best-first search would be a poor choice, since from one node to the next the problem could change substantially. However, when combined with the plunging heuristic, this same strategy becomes really interesting. Indeed, the plunging phase can be completely decoupled from the B&B. The CPU, acting as a decision maker, chooses a promising node and spawns a task that

will independently explore the subtree of this node. Once done, this task will report its findings to the decision maker. Moreover, when plunging, the next node to be solved differs by only the new bound on the branching variable. Communications are then minimized and the LP solver is usually able to quickly find the new solution since the problem has been only slightly altered.

Warmstart

There are two cases where starting from a fresh problem is required or beneficial. The first one is imposed by the numeric inaccuracy appearing after several iterations of the LP solver. The second is upon the request of a new subtree. To avoid the extensive communication costs of a full restart, the GPU keeps in memory an intermediate stable state of the problem, a *warmstart*. This state could, for example, be the one found after solving the root node of the tree.

Multi-GPU exploration

Having the CPU act as a decision maker and the GPU as an explorer, allows for the possibility of using multiple GPUs to explore the tree. The global knowledge is maintained by the CPU task. The CPU assigns to the GPUs the task of exploring subtrees of promising nodes. Since each plunging is done locally, no communications are required between the GPUs. Moreover, the amount of nodes processed during a plunging can be used to tune the load of the CPU task.

3. Performance model

Performance models are useful to predict the behaviour of implementations as a function of various parameters. Since the standard simplex algorithm is the easiest to understand, we will focus in this chapter on its behaviour as a function of the problem size.

CUDA kernels require a different kind of modeling than usually encountered in parallelism. The key idea is to capture in the model the decomposition into threads, warps, and blocks. One must also pay a particular attention to global memory accesses and to how the pipelines reduce the associated latency.

In order to model our implementation, we follow the approach given by K. Kothapalli et al. [19] (see also [17]). First, we examine the different levels of parallelism on a GPU. Then, we determine the amount of work done by a single task. By combining both analyses, we establish the kernel models. The final model then consists of the sum of each kernel.

This model has also been used to model a multi-GPUs version of the standard simplex method [2].

3.1. Levels of parallelism

A kernel can be decomposed into an initialization phase followed by a processing phase. During the initialization phase the kernel context is set up. Since this operation does not take a lot of time compared to the processing phase, it is needless to incorporate it into the model. The processing phase is more complex and its execution time depends directly on the amount of work it must perform. We shall now focus on this phase and on the various levels of parallelism on a GPU.

At the first level, the total work is broken down into components, the blocks. They are then dispatched on the available multiprocessors on the GPU. The execution time of a kernel depends on the number of blocks N_B per SM (streaming multiprocessor) and on the number N_{SM} of SM per GPU.

At a lower level, the work of a block is shared among the various cores of its dedicated SM. This is done by organizing the threads of the block into groups, the warps. A warp is a unit of threads that can be executed in parallel on an SM. The execution time of a block is then linked to the number N_W of warps per block, the number N_T of threads per warp, and the number N_P of cores per SM.

The third and lowest level of parallelism is a pipeline. This pipeline enables a pseudoparallel execution of the tasks forming a warp. The gain produced by this pipeline is expressed by its depth D.

3.2. Amount of work done by a thread

In the previous chapter, we defined the different levels of parallelism down to the smallest, namely the thread level. We must now estimate how much work is done by a single thread of a kernel in terms of cycles. It depends on the number and the type of instructions. In CUDA, each arithmetic instruction requires a different number of cycles. For example, a single precision add costs 4 cycles while a single precision division costs 36 cycles.

Moreover, since global memory access instructions are nonblocking operations, they must be counted separately from arithmetic instructions. The number of cycles involved in a global memory access amounts to a 4 cycle instruction (read/write) followed by a nonblocking latency of 400–600 cycles. To correctly estimate the work due to such accesses, one needs to sum only the latency that is not hidden. Two consecutive read instructions executed by a thread would cost twice the 4 cycles, but only once the latency due to its nonblocking behaviour. Once the amount of cycles involved in these memory accesses has been determined, it is then necessary to take into account the latency hidden by the scheduler (warp swapping).

The total work C_T done by a thread can be defined either as the sum or as the maximum of the memory access cycles and the arithmetic instructions cycles. Summing both types of cycles means we consider that latency cannot be used to hide arithmetic instructions. The maximum variant represents the opposite situation where arithmetic instructions and memory accesses are concurrent. Then only the biggest of the two represents the total work of a thread. This could occur for example when the latency is entirely hidden by the pipeline.

It is not trivial to define which scenario is appropriate for each kernel. There are many factors involved: the dependency on the instructions, the behaviour of the scheduler, the quantity of data to process, and so on.

If latency cannot be used to hide arithmetic instructions, the number of cycles C_T done by a thread can be defined as the sum of the global memory access and the arithmetic instructions. Otherwise, one must consider the maximum instead of the sum.

3.3. Global performance model

We now turn to the global performance model for a kernel. We must find out how many threads are run by a core of an SM. Recall that a kernel decomposes into blocks of threads, with each SM running N_B blocks, each block having N_W warps consisting of N_T threads. Also recall that an SM has N_P cores which execute batches of threads in a pipeline of depth D. Thus, the total work C_{core} done by a core is given by

$$C_{core} = N_B \cdot N_W \cdot N_T \cdot C_T \cdot \frac{1}{N_P \cdot D} \tag{3.1}$$

which represents the total work done by an SM divided by the number of cores per SM and by the depth of the pipeline. Finally, the execution time of a kernel is obtained by dividing C_{core} by the core frequency.

3.4. A kernel example: *steepest-edge*

The selection of the entering variable is done by the *steepest-edge* method as described in chapter 2.1.

The processing of a column is done in a single block. Each thread of a block has to compute $N_{el} = \frac{m}{N_W \cdot N_T}$ coefficients of the column. This first computation consists of N_{el} multiplications and additions. The resulting partial sum of squared variables must then be reduced on a single thread of the block. This requires $N_{red} = \log_2(N_W \cdot N_T)$ additions. Since the shared memory is used optimally, there are no added communications. Finally, the coefficient c_j must be divided by the result of the reduction.

Each block of the kernel computes $N_{col} = \frac{n}{N_B \cdot N_{SM}}$ columns, where N_{SM} is the number of SM per GPU. After processing a column, a block keeps only the minimum of its previously computed steepest-edge coefficients. Thus, the number of arithmetic instruction cycles for a given thread is given by

$$C_{Arithm} = N_{col} \cdot (N_{el} \cdot (C_{add} + C_{mult}) + N_{red} \cdot C_{add} + C_{div} + C_{cmp}) \tag{3.2}$$

where C_{ins}, with $ins \in \{add, div, mult, cmp\}$, denotes the number of cycles required by each instructions to execute.

Each thread has to load N_{el} variables to compute its partial sum of squared variables. The thread computing the division also loads the coefficient c_j. This must be done for the N_{col} columns with which a block has to deal. We must also take into account that the scheduler hides some latency by swapping the warps, so the total latency $C_{latency}$ must be divided by the number of warps N_W. Thus, the number of cycles relative to memory accesses is given by

$$C_{Accesses} = \frac{N_{col} \cdot (N_{el} + 1) \cdot C_{latency}}{N_W} \tag{3.3}$$

At the end of the execution of this kernel, each block stores in global memory its local minimum. The CPU will then have to retrieve the $N_B \cdot N_{SM}$ local minimums and reduce them. It is then profitable to minimize the number N_B of blocks per SM. With a maximum of two blocks per SM, the cost of this final operation done by the CPU can be neglected when m and n are big.

It now remains to either maximize or sum C_{Arithm} and $C_{Accesses}$ to obtain C_T. The result of equation (3.1) divided by the core frequency yields the time $T_{KSteepestEdge}$ of the steepest-edge kernel.

3.5. Standard simplex GPU implementation model

As seen in chapter 2.1, the standard simplex implementation requires only a few communications between the CPU and the GPU. Since all of these communications are constant and small, they will be neglected in the model. For the sake of simplicity, we will consider the second phase of the *two-phase simplex* where we apply iteratively the three main operations: selecting the entering variable, choosing the leaving variable and

pivoting. Each of these operations is computed as a kernel. The time of an iteration $T_{Kernels}$ then amounts to the sum of all three kernel times:

$$T_{Kernels} = T_{KSteepestEdge} + T_{KExpand} + T_{KPivoting} \tag{3.4}$$

The times $T_{KExpand}$ and $T_{KPivoting}$ for the expand and pivoting kernels are obtained in a similar way as for the steepest-edge kernel described in the previous chapter.

With the estimated time per iteration $T_{Kernels}$, we can express the total time T_{prob} required for solving a problem as

$$T_{prob} = T_{init} + r \cdot T_{Kernels} \tag{3.5}$$

where r is the number of iterations. Note that research by Dantzig [13] asserts that r is proportional to $m \log_2(n)$.

4. Measurements and analysis

Three sets of measurements are presented here. The first one is a validation of the performance model presented in chapter 3. The second is a comparison of the performances of the various implementations of the simplex method detailed in chapter 2. The third one is a study of the potential of the multi-GPU Branch-and-Bound method described in section 2.3.

As a preliminary, we ensured that our implementations were functional. For that matter, we used a set of problems available on the NETLIB Repository [15]. This dataset usually serves as a benchmark for LP solvers. It consists of a vast variety of real and specific problems for testing the stability and robustness of an implementation. For example, some of these represent real-life models of large refineries, industrial production/allocation, or fleet assignments problems. Our implementations are able to solve almost all of these problems.

4.1. Performance model validation

In order to check that our performance model for the standard simplex implementation is correct, a large range of problems of varying size is needed. As none of the existing datasets offered the desired diversity of problems, we used a problem generator. It is then possible to create problems of given size and density. Since usual LP problems have more variables than equations, our generated problems have a ratio of 5 variables for 1 equation ($n = 5 \cdot m$).

The test environment is composed of a CPU server (2 Intel Xeon X5570, 2.93GHz, with 24GB DDR3 RAM) and a GPU computing system (NVIDIA Tesla S1070) with the 3.2 CUDA Toolkit. This system connects 4 GPUs to the server. Each GPU has 4GB GDDR3 graphics memory and 30 streaming multiprocessors, each holding 8 cores (1.4GHz).

We validated our performance model by showing that it accurately fits our measurements. The correlation between the measurements and the model is above 0.999 (see Figure 4.1).

4.2. Performance study of our Simplex implementations

Four different implementations are compared in this chapter. We will refer to each of them according to the terminology introduced below.

- Standard simplex method improved ($\mathcal{O}(2mn)$): *Standard*

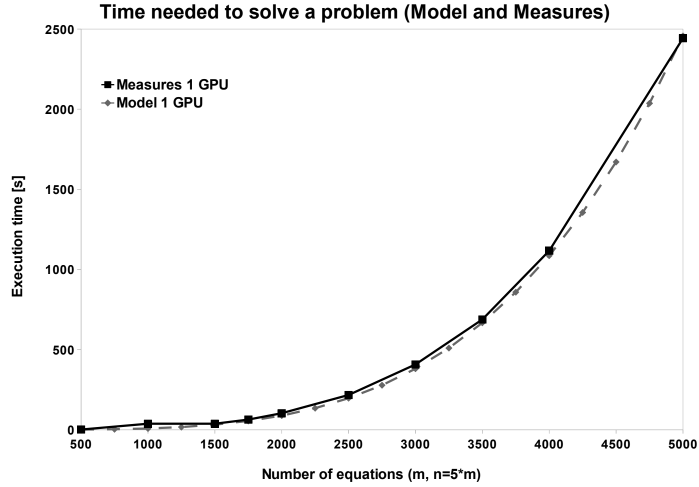


Figure 4.1.: Performance model and measurements comparison.

- Revised simplex method using basis kernel
  - without improvements ($\mathcal{O}(3mn + 4m^2)$): *Revised-base*
  - optimized ($\mathcal{O}(2mn + 3m^2)$): *Revised-opti*
  - optimized with sparse matrix-vector multiplication ($\mathcal{O}(2\theta + 3m^2)$): *Revised-sparse*

These implementations all use the *steepest-edge* and *expand* methods for the choice of, respectively, the entering and the leaving variables.

We used problems from the NETLIB Repository to illustrate the improvements discussed in chapter 2. The results expected from the first three methods are quite clear. The *Standard* method should be the best, followed by the *Revised-opti*, and then the *Revised-base*. However, the performance of the *Revised-sparse* implementation remains unclear since the value of θ is unknown and depends on the density and size of the constraints matrix. This is the main question we shall try to answer with our experiments.

The test environnement is composed of a CPU server (2 Intel Xeon X5650, 2.67GHz, with 96GB DDR3 RAM) and a GPU computing system (NVIDIA Tesla M2090) with the 4.2 CUDA Toolkit. This system connects 4 GPUs to the server. Each GPU has 6GB GDDR5 graphics memory and 512 cores (1.3GHz).

Let us begin by discussing the performances on problems solved in less than one second. The name, size, number of nonzero elements (NNZ), and columns to rows ratio (C-to-R) of each problem are reported in Table 4.1. The performances shown in Figure 4.2 corroborate our previous observations. On these problems the *Revised-sparse* method doesn't outperform the *Standard* one. This can be explained by two factors: the added communications (kernel calls) for the revised method, and the small size and density of the problems. It is likely that sparse operations on a GPU require larger amounts of

Problem Name	Rows	Cols	NNZ	C-to-R
etamacro.mps	401	688	2489	1.7
fffff800.mps	525	854	6235	1.6
finnis.mps	498	614	2714	1.2
gfrd-pnc.mps	617	1092	3467	1.8
grow15.mps	301	645	5665	2.1
grow22.mps	441	946	8318	2.1
scagr25.mps	472	500	2029	1.1

Table 4.1.: NETLIB problems solved in less than 1 second.

data to become more efficient than their dense counterparts.

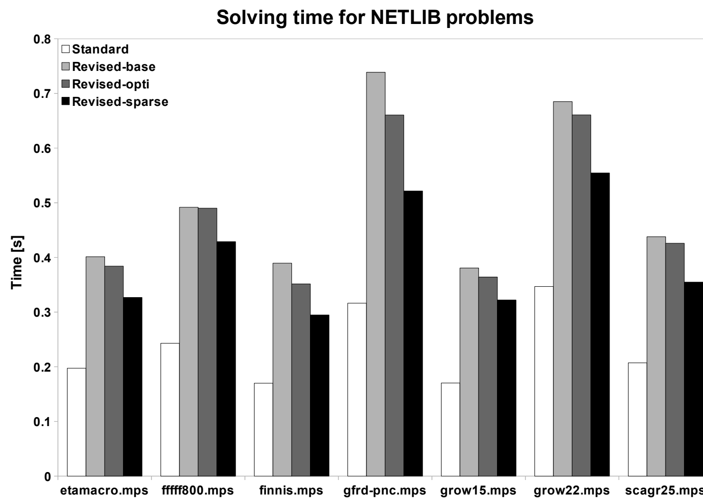


Figure 4.2.: Time required to solve problems of Table 4.1.

The problems shown in Table 4.2 are solved in less than 4 seconds. As we can see in Figure 4.3, the expected trend for the *Revised-base* and the *Revised-opti* methods is now confirmed. Let us presently focus on the *Standard* and *Revised-sparse* methods. Some of the problems, in particular *czprob.mps* and *nesm.mps*, are solved in a comparable amount of time. The performance gain of the *Revised-sparse* is related to the C-to-R ratio of these problems, displaying, respectively, a 3.8 and a 4.4 ratio.

Finally, the biggest problems, and slowest to solve, are given in Table 4.3. A new tendency can be observed in Figure 4.4. The *Revised-sparse* method is the fastest on most of the problems. The performances are still close between the best two methods on problems having a C-to-R ratio close to 2 such as bnl2.mps, pilot.mps, or greenbeb.mps. However, when this ratio is greater, the *Revised-sparse* can be nearly twice as fast as the standard method. This is noticeable on 80bau3b.mps (4.5) and fit2p.mps (4.3).

Problem Name	Rows	Cols	NNZ	C-to-R
25fv47.mps	822	1571	11127	1.9
bnl1.mps	644	1175	6129	1.8
cycle.mps	1904	2857	21322	1.5
czprob.mps	930	3523	14173	3.8
ganges.mps	1310	1681	7021	1.2
nesm.mps	663	2923	13988	4.4
maros.mps	847	1443	10006	1.7
perold.mps	626	1376	6026	1.0

Table 4.2.: NETLIB problems solved in the range of 1 to 4 seconds

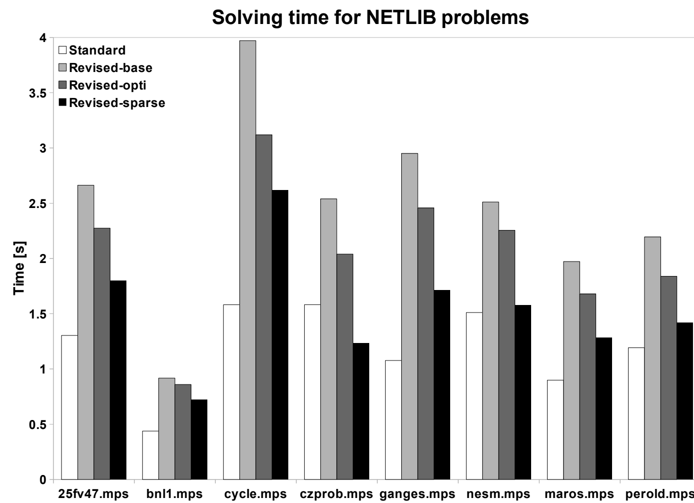


Figure 4.3.: Time required to solve problems of Table 4.2.

Although the C-to-R ratio of d6cube.mps (14.9) exceeds the ones previously mentioned, the *Revised-sparse* method doesn't show an impressive performance, probably due to the small amount of rows and the density of this problem, which doesn't fully benefit from the lower complexity of sparse operations.

4.3. Performance study of our parallel Branch-and-Bound implementation

In this section, we examine on what problem size our GPU implementation of the Branch-and-Bound (B&B) algorithm will become more efficient than a CPU-based counterpart. If we assume that in both cases the B&B strategy employed by the CPU is the same, the comparison can be made at the level of the LP solvers. Indeed, in our implementation

Problem Name	Rows	Cols	NNZ	C-to-R
80bau3b.mps	2263	9799	29063	4.3
bnl2.mps	2325	3489	16124	1.5
d2q06c.mps	2172	5167	35674	2.4
d6cube.mps	416	6184	43888	14.9
fit2p.mps	3001	13525	60784	4.5
greenbeb.mps	2393	5405	31499	2.3
maros-r7.mps	3137	9408	151120	3.0
pilot.mps	1442	3652	43220	2.5
pilot87.mps	2031	4883	73804	2.4

Table 4.3.: NETLIB problems solved in more than 5 seconds.

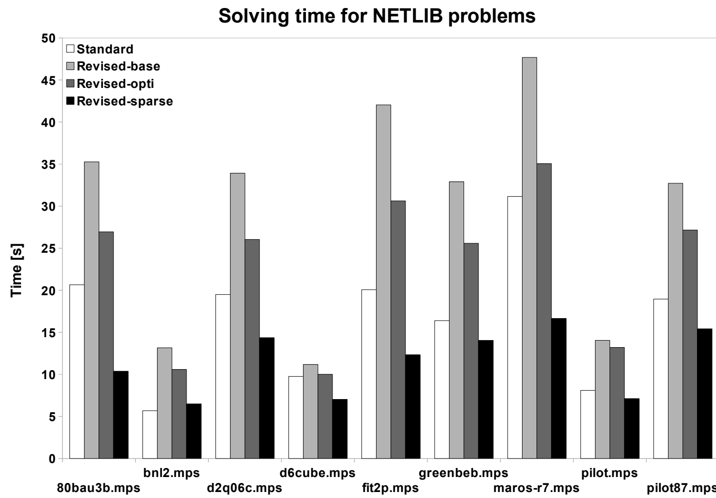


Figure 4.4.: Time required to solve problems of Table 4.3.

of the B&B algorithm, multiple LP solvers (in our case revised simplexes) run on several GPUs, this without any inter-GPU communications. With respect to node selection, the depth of the plunging can be chosen so that CPU-GPU communications are negligeable compared to the time to process the nodes during a plunging. We will use as a reference one of the best open-source LP solvers, namely the CLP solver from the COIN-OR project.

Since GPUs are more efficient when running on big data-parallel compute-intensive applications, we require a large range of problems of varying size. Towards this end, we used a random problem generator that enable the creation of problems with given size and density (*i.e.* the ratio of non-zero elements in the constraints matrix). Since usual LP problems have more variables than equations, our generated problems have a ratio of 5 variables for 1 equation. It is to be noted that their structure might not be like a

real-world one.

Our test environnement is composed of a CPU server (2 Intel Xeon X5650, 2.67GHz, with 96GB DDR3 RAM) and a GPU computing system (NVIDIA tesla M2090) with the 4.2 CUDA Toolkit. This system connects 4 GPUs to the server. Each GPU has 6GB GDDR5 graphics memory and 512 cores (1.3GHz). Such a GPU offers at peak performance up to 1.3Tflops for single precision floating point, 665Gflops for double precision, and 177GB/s memory bandwidth between threads (registers) and global memory.

We measured execution times of our GPU-based revised simplex implementation and of CLP on problems of varying size and of density 0.1%, 0.9% and 1.8%. Real-life models typically have densities lower than 1%. The results are shown in figure 4.5 where a Log-Log scale is used (natural logarithm).

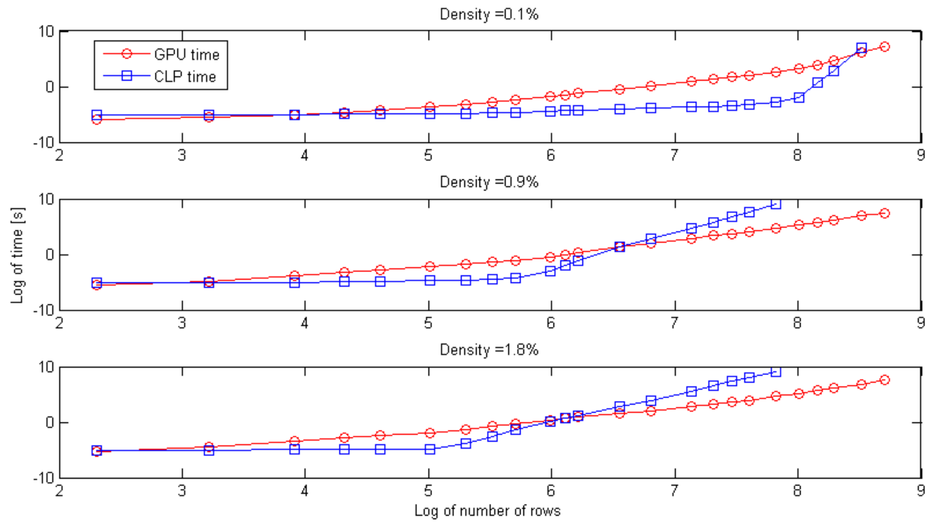


Figure 4.5.: Execution times of our revised simplex GPU implementation and the CLP solver on randomly generated problems of various sizes and densities of 0.1%, 0.9% and 1.8%. The number of columns is five times the number of rows. A Log-Log scale is used.

We can observe that, as density increases, execution times increase in proportion for a given problem size. Indeed, the sparse data structure implies that the execution times essentially depend on the number of non-zero elements in the constraints matrix.

A GPU becomes fully efficient when its computing units are all concurrently in use. This requires that the amount of data processed is sufficient. The problem size at which our GPU implementation becomes better than CLP diminishes as the density increases. This is probably due to the fact that CLP is designed to perform better on sparse problems. Consequently, our revised simplex GPU implementation will only outperform CLP on very large problems. One should note that beyond a certain amount of time, we didn't record the execution time of CLP, since this was of no practical use.

In table 4.4 are reported the timings measured on such problems available in the

NETLIB [15] repository. As mentioned previously, this benchmark represents a wide variety of real and specific problems that can be used to test the stability and robustness of an implementation. For example, some of these represent real-life models of large refineries, industrial production/allocation or fleet assignments problems. Given the small size and the high sparsity of these problems, as expected in this range, CLP is an order of magnitude faster than our GPU implementation. However, on big problems with high density of this dataset, such as `pilot`, `pilot87` or `d6cube`, our implementation performance was closer to the one of CLP with a slowdown factor of approximately 2.5.

Problem name	Rows	Cols	density[%]	GPU[s]	CLP[s]
etamacro.mps	401	688	0.9	0.33	0.028
fffff800.mps	525	854	1.4	0.47	0.031
finnis.mps	498	614	0.9	0.3	0.020
gfrd-pnc.mps	617	1092	0.5	0.54	0.026
grow15.mps	301	645	2.9	0.35	0.040
grow22.mps	441	946	2.0	0.59	0.060
scagr25.mps	472	500	0.9	0.37	0.026
25fv47.mps	822	1571	0.9	1.80	0.304
bnl1.mps	644	1175	0.8	0.74	0.088
cycle.mps	1904	2857	0.4	2.33	0.13
czprob.mps	930	3523	0.4	1.25	0.061
ganges.mps	1310	1681	0.3	1.77	0.029
nesm.mps	663	2923	0.7	1.54	0.217
maros.mps	847	1443	0.8	1.30	0.071
perold.mps	626	1376	0.7	1.45	0.186
80bau3b.mps	2263	9799	0.1	10.6	0.416
bnl2.mps	2325	3489	0.2	6.66	0.256
d2q06c.mps	2172	5167	0.3	15.02	2.06
d6cube.mps	416	6184	1.7	6.22	2.72
fit2p.mps	3001	13525	0.1	63.0	1.72
greenbeb.mps	2393	5405	0.2	14.26	0.886
maros-r7.mps	3137	9408	0.5	17.43	2.02
pilot.mps	1442	3652	0.8	7.32	3.08
pilot87.mps	2031	4883	0.7	15.77	7.34

Table 4.4.: Execution times on real-world problems from the NETLIB repository. These problems are of small size.

5. Conclusion and perspectives

In this chapter, we have presented the knowledge and understanding necessary in our view to produce an efficient integer linear programming solver on a GPU. We proposed various solutions to implement the standard and revised simplex. We have discussed the main concepts behind a branch-and-bound algorithm and pointed out some major concerns when it is coupled with a GPU solver. The results obtained with the standard simplex implementation allowed us to validate a detailed performance model we had proposed. Finally, we used problems from the NETLIB library to compare the performances of our various implementations. The revised simplex implementation with sparse matrix operations showed the best performances on time-consuming problems, while the standard simplex one was more competitive on easier problems. However, sequential open-source solvers such as CLP of the COIN-OR project still outperform such GPU implementations on small to mid-sized and highly sparse problems. We then used a problem generator in order to establish a size and sparsity limit above which this tendency would tip.

We shall now discuss some remaining potential improvements. A first step towards performance would be to consider the dual revised simplex [33]. While being similar to the methods treated in this chapter, it has the capacity to greatly reduce the time needed to solve problems. Yet, the main improvement would be seen when tackling larger problems than the ones considered here. Indeed, problems having hundreds of thousands of variables would technically be solvable on GPU devices with 2GB to 4GB of global memory. Moreover, such amounts of data would fully benefit from the potential of these devices. However, solving problems of this size raises an issue not addressed here: numerical instabilities. This phenomenon is due to the limited precision of mathematical operations on computing devices. Let us illustrate this problem on the inverse $\mathbf{B^{-1}}$ of the basis matrix $\mathbf{B}$. At each iteration of the revised simplex, $\mathbf{B^{-1}}$ is updated from its previous values. Performing this update adds a small error to the result. Unfortunately, these errors accumulate at each iteration, bringing at some point the $\mathbf{B^{-1}}$ matrix into a degenerate state. To avoid this situation, the matrix $\mathbf{B^{-1}}$ must be recomputed afresh once in a while by inverting the matrix $\mathbf{B}$.

Instead of directly processing the inverse of the matrix $\mathbf{B}$, it is more common to use some specific arithmetical treatment. Most simplex implementations use the so-called LU decomposition of $\mathbf{B}$ as a product of a lower and an upper triangular matrix [3]. Since the matrix $\mathbf{B}$ is sparse, the sparse version of this decomposition can be used and the corresponding update performed for $\mathbf{B^{-1}}$ at each iteration [28]. The sparse LU decomposition on CPU has been recently the subject of a large amount of research, yielding many improvements [27]. Once its GPU counterpart is available, this might result in improved and competitive simplex implementations on GPU.

To conclude, while the presented methodology should enable efficient ILP solver on GPU, we propose two trends to further improve our hybrid CPU-GPU integer linear programming solver: (1) assign to the CPU additional tasks such as generating cutting-planes or performing pre-analyses on the problem; (2) further optimize the GPU implementation of the LP solver.

Bibliography

[1] T. Achterberg. *Constraint integer programming.* PhD thesis, TU Berlin, 2007.

[2] X. Meyer, P. Albuquerque, and B. Chopard. A multi-GPU implementation and performance model for the standard simplex method. http://spc.unige.ch/lib/exe/fetch.php?media=pub:sgpu_europar2011.pdf. Presented at the 1st Int'l Symp. and 10th Balkan Conf. on Operational Research, BalcOR, Thessaloniki, Greece, Sept. 22-24, 2011.

[3] R. H. Bartels and G. H. Golub. The simplex method of linear programming using LU decomposition. *Commun. ACM*, 12(5):266–268, May 1969.

[4] J. Bieling, P. Peschlow, and P. Martini. An Efficient GPU Implementation of the Revised Simplex Method. In *Proc. of the 24th Int'l Parallel and Distributed Processing Symp.* IEEE, 2010.

[5] Robert E Bixby. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica*, pages 107–121, 2012.

[6] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67, May 2011.

[7] A. Boukedjar, M. E. Lalami, and D. El-Baz. Parallel Branch-and-Bound on a CPU-GPU System. In Rainer Stotzka, Michael Schiffers, and Yannis Cotronis, editors, *PDP*, pages 392–398. IEEE, 2012.

[8] T. Carneiro et al. A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU. In *23rd Int'l Symposium on Computer Architecture and High Performance Computing*, pages 41–47, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[9] L. G. Casado et al. Branch-and-Bound interval global optimization on shared memory multiprocessors. *Optimization Methods Software*, 23(5):689–701, October 2008.

[10] I. Chakroun et al. Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. *Concurrency and Computation: Practice and Experience*, 2012.

[11] V. Chvatal. *Linear Programming.* W.H. Freeman, New-York, 1983.

[12] Barry A Cipra. The best of the 20th century: editors name top 10 algorithms. *SIAM news*, 33(4):1–2, 2000.

[13] G. B. Dantzig. Expected number of steps of the simplex method for a linear program with a convexity constraint. Technical report, Stanford University, 1980.

[14] G. B. Dantzig and W. Orchard-Hays. *The Product Form for the Inverse in the Simplex Method*. Defense Technical Information Center, Santa Monica, California, 1953.

[15] J. Dongarra and E. Grosse. The NETLIB Repository at UTK and ORNL. http://www.netlib.org.

[16] M. E. Lalami, D. El-Baz, and V. Boyer. Multi-GPU implementation of the simplex algorithm. *IEEE International Conference on High Performance Computing and Communications*, pages 179–186, 2011.

[17] D. G. Spampinato, A. C. Elster, and T. Natvig. Modeling multi-GPU systems in parallel computing: From multicores and GPU's to petascale. *Advances in Parallel Computing*, 19:562–569, 2010.

[18] J. Bieling et al. An efficient GPU implementation of the revised simplex method. In *Proc. of the 24th Int'l Parallel and Distributed Processing Symp.* IEEE, 2010.

[19] K. Kothapalli et al. A performance prediction model for the CUDA GPGPU platform. Technical report, Int'l Inst. of Information Technology, Hyderabad, 2009.

[20] M. Benichou et al. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1):76–94, 1971.

[21] P. E. Gill et al. A pratical anti-cycling procedure for linearly constrained optimization. *Mathematical Programming*, 45:437–474, 1989.

[22] J. J. H. Forrest, J. P. H. Hirst, and J. A. Tomlin. Practical solution of large mixed integer programming problems with umpire. *Management Science*, 20(5):736–773, 1974.

[23] B. Gendron and T. G. Crainic. Parallel Branch-and-Bound Algorithms: Survey and Synthesis. *Operations Research*, 42:1042–66, 1994.

[24] D. Goldfarb and J. K. Reid. A practicable steepest-edge simplex algorithm. *Mathematical Programming*, 12:361–371, 1977.

[25] G. Greeff. The revised simplex algorithm on a GPU. Technical report, Department of Computer Science, Stellenbosch University, Stellenbosch, South Africa, February 2005.

[26] H. J. Greenberg (editor). *Tutorials on Emerging Methodologies and Applications in Operations Research*, volume 76 of *International Series in Operations Research & Management Science*. Springer, 2005.

[27] L. Grigori, J. Demmel, and H. Xiang. CALU: A Communication Optimal LU Factorization Algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32:1317–1350, 2011.

[28] R. K. Brayton, F. G. Gustavson, and R. A. Willoughby. Some results on sparse matrices. *Mathematics of Computation*, 24(112):937–954, 1970.

[29] J. A. J. Hall. Towards a practical parallelisation of the simplex method. *Computational Management Science*, 7(2):139–170, 2010.

[30] Karla L Hoffman and Manfred Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39(6):657–682, 1993.

[31] J. H. Jung and D. P. O'Leary. Implementing an interior point method for linear programs on a CPU-GPU system. *Electronic Transactions on Numerical Analysis*, 28:174–189, 2008.

[32] T. Achterberg, T. Koch and A. Martin. Branching rules revisited. *Oper. Res. Lett.*, 33(1):42–54, January 2005.

[33] C. E. Lemke. The dual method of solving the linear programming problem. *Naval Research Logistics Quarterly*, 1(1):36–47, 1954.

[34] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11:173–187, 1997.

[35] H. Marchand, A. Martin, R. Weismantel, and L. Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Appl. Math.*, 123(1-3):397–446, November 2002.

[36] M.-S. Mezmaz, N. Melab, and T. El-Ghazali. A Grid-enabled Branch-and-Bound Algorithm for Solving Challenging Combinatorial Optimization Problems. In *21th Int'l Parallel and Distributed Processing Symp., IPDPS*, pages 1–9. IEEE, 2007.

[37] Sparsh Mittal and Jeffrey S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.*, 47:69:1–69:35, July 2015.

[38] Hans Mittelmann. Benchmarks for optimization software, 2015.

[39] G. Sierksma. *Linear and Integer Programming: Theory and Practice, Second Edition.* Chapman & Hall/CRC Pure and Applied Mathematics. Taylor & Francis, 2001.

[40] D. G. Spampinato, A. C. Elster, and T. Natvig. Modeling Multi-GPU Systems in Parallel Computing: From Multicores and GPU's to Petascale. *Advances in Parallel Computing*, 19:562–569, 2010.

[41] George J. Stigler. The Cost of Subsistence. *Journal of Farm Economics*, 27(2):303–314, May 1945.

[42] L. M. Suhl and U. H. Suhl. A fast LU update for linear programming. *Annals of Operations Research*, 43:33–47, 1993.

[43] A. Swietanowski. A new steepest edge approximation for the simplex method for linear programming. *Computat. Optimization and Appl.*, 10:271–281, 1998.

[44] M. E. Thomadakis and J.-C. Liu. An efficient steepest-edge simplex algorithm for SIMD computers, 1996.

[45] V. Volkov. Better performance at lower occupancy. http://www.eecs.berkeley.edu/∼volkov. Presented at the GPU Technology Conference 2010 (GTC 2010), San José, California, USA.

[46] K. Wolter. *Implementation of cutting plane separators for mixed integer programs.* PhD thesis, TU Berlin, 2006.

Part II.

Fluid simulation in the hearing organ

Preamble

Georg Von Békésy was awarded a nobel price in 1961 for his pioneering work on the cochlea function in the mammalian hearing organ [2]. He postulated that the placement of sensory cells in the cochlea corresponds to a specific frequency of sound. This theory, known as tonotopy, is the ground of our understanding on this complex organ. With the advance of technologies, this knowledge broaden continuously and seems to confirm Békésy initial observations. However, a mystery still lies in the center of this organ : how does its microscopic tissues exactly act together to decode the sounds that we perceive ?

One of these tissues, the Reissner membrane, forms a double cell layer elastic barrier separating two fundamental ducts of this organ. Yet, until recently [15], this membrane, was not considered in the modelling of the inner ear due to its smallness. Nowadays, objects of this size are at the reach of the medical imagining and measuring expertise [17, 4, 15]. Newly available observations coupled with the increasing availability of computational resources should enable modellers to consider the impact of these microscopic tissues in the inner ear mechanism.

In this thesis part, we explore the potential fluid-structure interactions happening in the inner ear, more particularly on the Reissner membrane. This study aims at answering two separate questions :

- Can nowadays computational fluid dynamics solvers simulate interaction with inner ear microscopic tissues ?
- Has the Reissner membrane function on the auditory system been overlooked ?

This part is organized as follow. Starting with a brief state of the art, the first chapter introduces the required notions to understand the experiments. The anatomy and the function of the cochlea are summarized and the main concepts of the computational fluid dynamics method used are defined. The next two chapters presents the setting of the simulations and their results : the first focuses on the vestibular duct and the Reissner membrane while the second focuses on the cochlear duct and the organ of Corti. We conclude in the final chapter by discussing the numerical experiments results.

1. Introduction

1.1. State of the art

Nowadays, more than 60'000 hearing impaired persons benefits from cochlear implants [24]. These devices design is strongly tied to our understanding of the hearing organ. Its main component, the cochlea, processes acoustic signals by the mean of a complex mechanism. Since the early work of Georg Von Békésy [1, 2], research on the mammal cochlea has been the focus of many scientists [20, 16]. Yet, this mechanism is not fully understood.

In order to fill this knowledge gap, a large amount of cochlear models have been developed through the years [14]. Given the complexity of the hearing organ, these models focus on a set of selected phenomenon : cochlear micro- or macro-mechanics with or without fluid coupling. Macro-mechanics models represent roughly the cochlea as a box with the vestibular and tympanic ducts separated by the vibrating basilar membrane [5, 13, 26]. Micro-mechanics models focus on reproducing more accurately the organ of Corti [3, 7, 18]. While all these models vary in the methods used and the size of phenomena simulated, they all start with the same assumption that the Reissner membrane does not play any major role in the mechanics of the cochlea [14]. However, this membrane, that separates two fundamental ducts of the cochlea, has been shown to propagate travelling waves in a comparable fashion as the widely considered basilar membrane [15].

To our knowledge, computational fluid dynamics (CFD) has only been used in one model [8]. This model presented an analysis of the macro-mechanics of the cochlea using immersed boundary conditions : a method simulating solid structure immersed in fluids. Our work is based on a similar simulation method but differs strongly in its aim. Our focus is the under-considered deformation of the Reissner membrane and its induced fluid displacement. We postulate that such fluid movements could impact the microscopic elements of Corti's organ and therefore play a key role in the hearing mechanism. In order to tackle this challenge of representing both the inner ear micro- and macro-mechanics in the same simulation setting, we use a highly-parallel CFD code, Palabos[1].

[1] http://www.palabos.org/

1.2. Cochlea : the hearing organ

1.2.1. Anatomy

The inner ear is composed of two major organ linked by the vestibule : the vestibular system and the cochlea (Fig. 1.1B). The main function of the first one is the sense of balance, while the second contains the primary auditory organ of the inner ear [23].

The cochlea whose name comes from the ancient greek *kohlias*, meaning spiral or snail shell, is structured as a spiral-shaped cavity. This cavity is considered to start from the base in the middle of the vestibule and continues until it reaches its apex, the helicotrema. In humans, this canal forms in average 2.5 turns over 35mm and is separated by a thin spiral shelf of bone, the osseous spiral lamina (Fig. 1.2A).

The cochlea can further be decomposed in three ducts separated by two membranes : the basilar and the Reissner membranes (Fig. 1.2B). One of these ducts, the vestibular duct, or scala vestibuli, starts from the vestibule and ends at the apex of the cochlea. The second duct, the tympanic duct, or scala tympani, starts from the tympanic cavity and connects with the vestibular duct at the cochlea apex through a small opening, the helicotrema. Both ducts contains a fluid, the perilymph that is sealed, on the opposite side of the helicotrema, by the oval window in the vestibule and the round window in the tympanic cavity.

In between both these ducts lies the cochlear duct or scala media. This duct contains a fluid, the endolymph and is separated, on one side, from the scala vestibuli by the delicate two-cells thick Reissner membrane. On the other side, it is separated from the scala tympani by the elastic basilar membrane. This membrane thickness reduces progressively from the base to the apex of the cochlea.

Inside the scala media, on top of the basilar membrane rests the small yet important organ of Corti (Fig. 1.3). This complex organ contains the fundamental element that translates the sound vibrations into nervous signals : the inner ear hair cells. These cells stand in a small canal, closed on one side by the basilar membrane and on the other side by the tectorial membrane.

1.2.2. The hearing mechanism

Acoustic signals are conveyed in the inner ear by a vibratory pattern. Acoustic waves reach the external auditory canal, the tympanic membrane, and transmit vibrations through the ossicles, a complex of small bones (malleus, incus, stapes). These bones amplify the vibratory signal before applying it to the oval windows (Fig. 1.1A[2]). In turn, this membranous window transmits the vibrations to the vestibule and to the inner ear fluid. These vibrations in the fluids are considered to create a travelling wave on the basilar membrane. The wave evolve from the base toward the apex for a length inversely proportional to its frequency. Therefore the placement, along the cochlea, of inner ear hair cells defines the sound frequencies that activate them.

[2] The tympanic membrane or eardrum is represented in red and the ossicles in green.

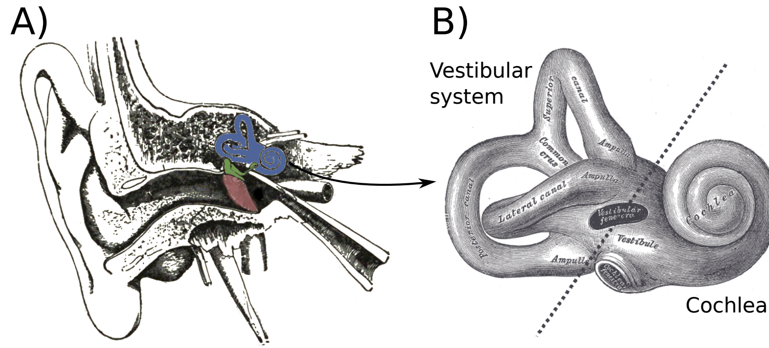


Figure 1.1.: Figure A) shows the chain of transmission of sound with the tympanic membrane (red), the ossicles (green) and the inner ear (blue). Figure B) shows the inner ear with its two main components : the vestibular system and the cochlea.

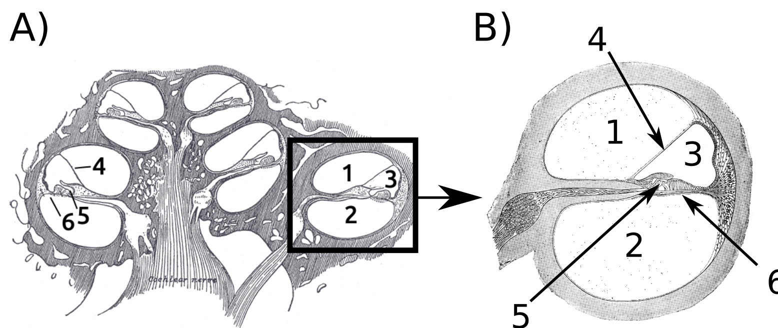


Figure 1.2.: Figure A) shows a cut of the cochlea and figure B) shows an enlargement of one cross-section of the ducts. The following elements are annotated : 1) Scala vestibuli; 2) Scala tympani; 3) Scala media; 4) Reissner membrane; 5) Corti's organ; 6) Basilar membrane.

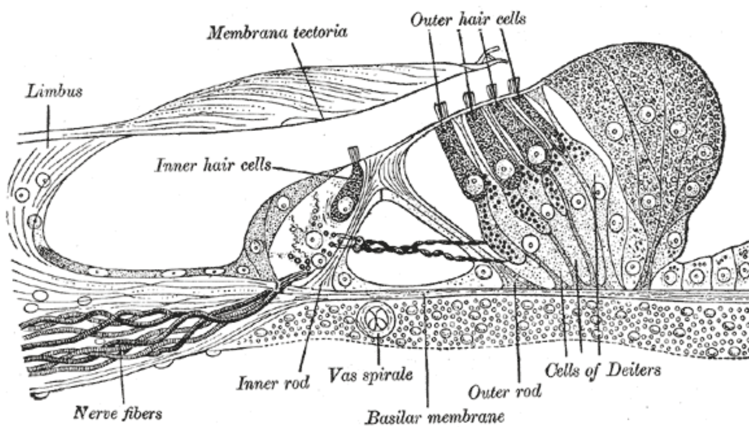


Figure 1.3.: Organ of Corti.

This theory, more known as tonotopy, was proposed by Georg Von Békésy [1, 2] and is based on his observations of the basilar membrane vibratory response when excited by sounds. This explanation of the hearing mechanism grossly simplify the reality by hiding complex micro-mechanical phenomenon occurring in the organ of Corti. Moreover, it doesn't account of the role of the Reissner membrane. More complex hypothesis are still explored in order to explain the impact of the Reissner membrane [15] or the role of the cochlea active mechanism that produce otoacoustic emissions [10].

1.3. Computational fluid simulations

1.3.1. Palabos, an open source highly parallel solver

In order to simulate the complex fluid-solid phenomenons occurring in the inner ear, we used the lattice Boltzmann method (LBM). This method is a modern approach in Computational Fluid Dynamics. It is often used to solve the incompressible, time-dependent Navier-Stokes equations numerically. Its strength lies in the ability to easily represent complex physical phenomena, ranging from multiphase flows to chemical interactions between the fluid and the surroundings. The method finds its origin in a molecular description of a fluid and can directly incorporate physical terms stemming from a knowledge of the interaction between molecules.

Unlike traditional CFD methods, LBM models the fluid consisting of fictive particles, and such particles perform consecutive propagation and collision processes over a discrete lattice mesh. Due to its particulate nature and local dynamics, LBM has several advantages over other conventional CFD methods, especially in dealing with complex boundaries, incorporating microscopic interactions, and parallelization of the algorithm[3]. For that matter, we chose ot use the open-source software Palabos[4]. This software is a massively-parallel Lattice Boltzmann implementation that have been used in numerous applications such as the simulation of blood cells deposition in aneurysms [25] or permeability change of porous medium [9].

In order to simulate membranes, Palabos maintainers developed an elastic shell model. In this model, the fluid domain is bounded by time-independent boundaries (rigid walls) and time-dependent boundaries (membranes). During the simulation, the fluid-structure interaction is taken in account at these boundary such as to apply internal forces on the elastic wall. The time dynamics of the wall is then solved and the fluid domain is recomputed to adapt to these changes. This model implement all these steps as to be second-order accurate and offers a good scaling for parallel execution.

1.3.2. Palabos elastic shell model

Time dependent boundaries, or membrane, are represented by triangular surfaces in a three dimensional space (Fig. 1.4). In order to simulate membrane movement, the

[3] http://en.wikipedia.org/wiki/Lattice_Boltzmann_methods
[4] http://www.palabos.org/

surface vertices or membrane particles P are displaced according to the stress generated by the fluid. This mechanism is guided by the simplified following iterative algorithm :

1. Fluid particles inside the fluid domain are detected and updated using the lattice-Boltzmann algorithm.
2. The fluid stresses on wall are computed and extrapolated to the membrane particles.
3. The elastic stress at each of these particles is then computed.
4. Membrane particles advance according to Newton's law.
5. The off-lattice boundary condition is reconstructed according to the new membrane position.

The elasticity of the membrane is characterized by a complex model formed of stretching, bending and shearing forces (Fig. 1.4). The first force is modelled by considering the edges between vertex as spring with linear attractive forces. The second is insured by preserving the on-membrane angles. The latter is defined as the preservation of the triangular surfaces area.

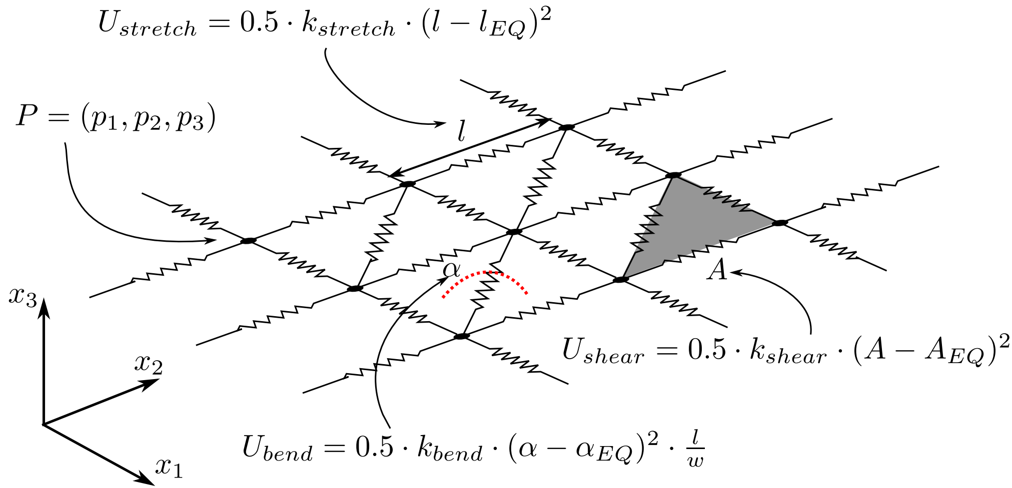


Figure 1.4.: Representation of a time-dependent boundary and the three forces acting on it : stretching, shearing and bending.

At each iteration, these elastic forces are computed and used to represent the acceleration of each membrane particles. The elastic potential of a particle U is represented by multiple potential in order to represent as realistically as possible the three dimensions.

The stretch potential exists between a particle P and each of its neighbours. This potential is given by

$$U_{stretch} = 0.5 \cdot k_{stretch} \cdot (l - l_{EQ})^2$$

where l stand for the distance between the particles at this iteration and l_{EQ} their distance at equilibrium.

The bending potential represent the angular elasticity of the membrane. For a given membrane particle, each pair of adjacent triangular surface generate a bending potential given by

$$U_{bend} = 0.5 \cdot k_{bend} \cdot (\alpha - \alpha_{EQ})^2 \cdot \frac{l}{w}$$

where α stands for the angle between both triangular surfaces, α_{EQ} the equilibrium angle, l represents the length of the shared edge. Then tile span w between both triangular surface is computed as $w = \frac{h}{6}$ with h being the summed height of the triangular surfaces.

The shear potential represent the elasticity of an area of membrane and exists for each triangular surface where the particle stand as a vertex.

$$U_{shear} = 0.5 \cdot k_{shear} \cdot (A - A_{EQ})^2$$

where A represent the area of triangular surface and A_{EQ} the area at equilibrium.

These three potentials are then summed in order to define the total elastic potential of a given particle

$$U = U_{stretch} + U_{shear} + U_{bend}$$

and from this total potential, the force F_U on a particle P is numerically derived over all three dimensions $X = (x_1, x_2, x_3)$

$$F_U = \frac{dU}{dX} = \frac{U(x + \epsilon) - U(x - \epsilon)}{2 \cdot \epsilon}$$

Another force is then added to this model in order to represent the friction caused by the membrane. This force decelerate a membrane particle by applying a negative force proportional to its velocity v

$$F_F = -k_{friction} \cdot v$$

Finally, Newton's law is applied to define the acceleration incurring on the particle

$$a = \frac{F_U + F_F}{m} = \frac{F_U + F_F}{\tau * A}$$

where τ is a constant representing the shell density and A the area around the particle.

In conclusion, we presented a model that has the power to simulate three dimensional elastic membranes interacting with fluid, for example the Reissner membrane and inner ear fluids. This model uses three forces to represent elasticity and takes into account the density, or thickness, of the membrane by two means : the friction force F_F and the shell density τ.

2. Simulation of the vestibular duct

The cochlea has a complex geometry made of multiple coiled ducts (Fig. 1.2). The cross section of each of these ducts varies from the base to the apex of the cochlea and are varying across species [4, 21, 23]. In order to better understand the various characteristics of these ducts, we start by a simplistic approximation of a duct as a simple canal and progressively make it evolve toward a more realistic cochlea duct.

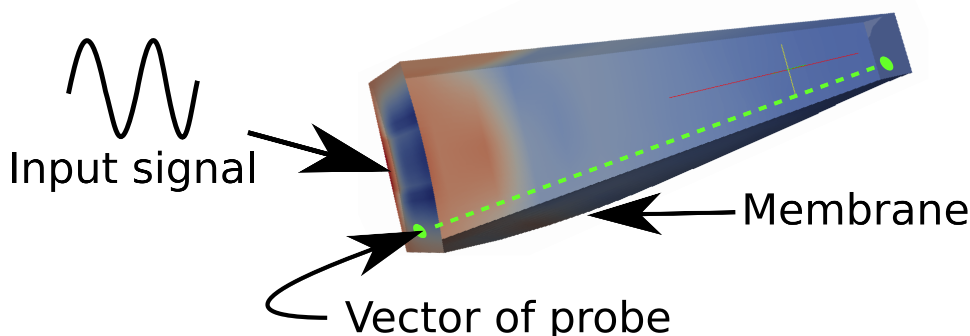


Figure 2.1.: Simulation setting for the vestibular duct. Periodic vibrations are applied to the frontal lid as input signal. A vector of probe lies in vicinity of the time-dependent boundary, the membrane, in order to measure the fluid velocity over time and membrane position.

We want to observe the deformation of the Reissner membrane induced by wave propagation travelling in the vestibular duct. We represented this duct as a simple rigid canal having two moving walls. One one end of the canal, the first moving wall, or membrane, is subject to periodic sinusoidal deformations (Fig. 2.1). These periodic deformations imitated the mechanical vibrations of the oval window. The bottom side of the duct defined then the second membrane representing the elastic properties of the Reissner membrane. The fluid interaction on the membrane was observed using a probe vector that measured the fluid velocity close to the membrane from base to apex (Fig. 2.1 in green).

2.1. Impact of the duct geometry

Duct geometry plays a fundamental role on the propagation of mechanical waves [11]. Waveguides dimension and structure limit the frequency range of transported waves. In addition to these limitations, the energy of these waves decrease in function of the waveguide boundary and the medium, or fluid in our case, in which they are propagated.

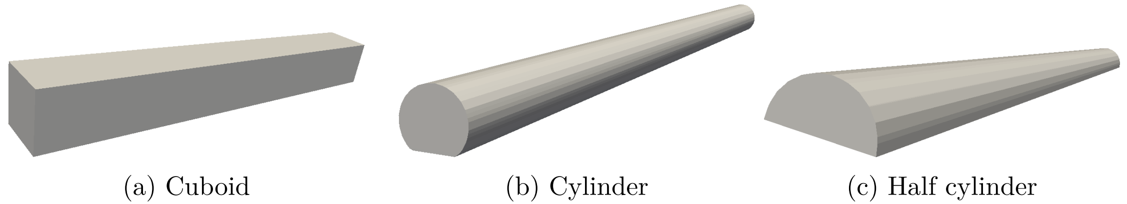
The fluid viscosity, heat conduction and internal molecular processes each may generate losses in the wave intensity.

In these experiments, we particularly focused on measuring the impact of the duct geometry on the velocity of travelling waves. Therefore, we did not consider a realistic setting in respect of the characteristic viscosities and frequencies of the human auditory system. We used geometries respecting the cross section to length ratio of the vestibular duct and then insured the simulation stability in order to monitor five different input vibrations (Fig. 2.1). Each vibrations had its own period w_k and their combination, $\sum_{k=1}^{5} \sin(w_k t)$, defined the imposed oval window deformation.

Simulation setting a) : We fixed the Reynolds number to one and the characteristic lattice velocity in lattice units to 0.01. The characteristic duct size, represented by the oval window size, was set to of 0.5 meters. Lattice units corresponded then to $\delta x = 10^{-2}m$, $\delta t = 10^{-2}s$ and a fluid cinematic viscosity of $5 \cdot 10^{-3}m^2/s$ close to the one of oil. Elastic constants of the Reissner membrane ($k_{stretch}, k_{shear}, k_{bend}$) were arbitrarily fixed to 0.05 in lattice unit. This setting required, per simulation, an average of 20 millions lattice site that were simulated over 10'000 iterations on 60 processing units for a total simulation time of one day.

2.1.1. Rectangular or cylindrical ducts

The aim of this first simulation is to measure the impact on wave absorption and attenuation of simplistic geometries. We are comparing three geometries represented in figure 2.2 : (a) rectangular, (b) cylindrical and (c) half-cylindrical. Each of their flat bottom side is a time-dependent boundary representing the Reissner membrane.



(a) Cuboid (b) Cylinder (c) Half cylinder

Figure 2.2.: Geometries compared.

For each of these geometries, we applied vibrations at five different frequencies (0.24, 0.33, 0.45, 0.67 and 1.0 hertz) on the duct lid. We then measured the fluid velocity close to the membrane and thus obtained temporal measures of velocity for 1200 points along the membrane. For each spatial points, we computed the frequency spectrum of the fluid velocity over time using a discrete Fourier transform. The single-sided amplitude spectrum[1] was then extracted and represented against the position in the membrane and the frequency (Fig. 2.3).

The five input frequencies are well identifiable for all three geometries. In all of them, the velocity of waves with higher frequencies decrease faster along the membrane than

[1] The single-sided amplitude spectrum represents the sum of the absolute amplitude of negative and positive frequencies.

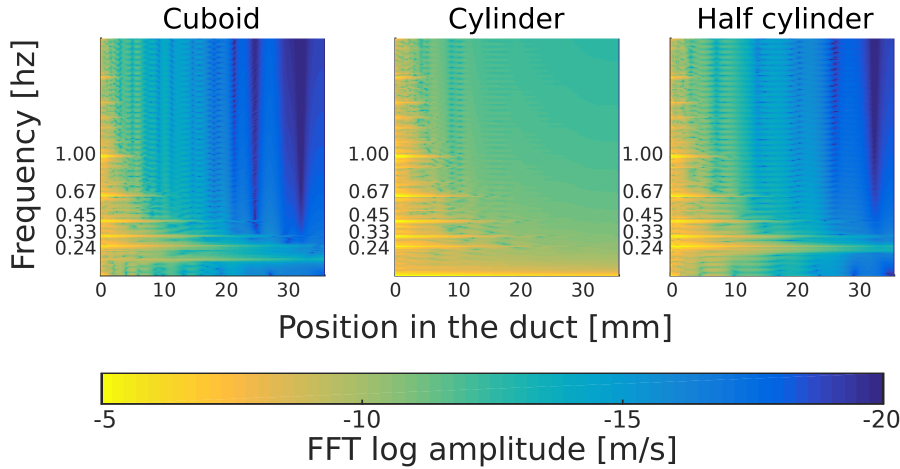


Figure 2.3.: Overview of the single-sided spectrum amplitude of the velocity along the membrane. X axis represent the position in the duct and Y axis the frequency. The colour indicate the value of the log single-sided spectrum amplitude of the velocity.

waves with low frequencies. For each of the imposed vibrations, multiple harmonics appear. However, their weaker velocity decay in the first third of the duct.

This overview shows that these three geometries propagate waves differently. While the same waves frequency are observed, their amplitude along the membrane differ. We postulate that the cuboid absorb more wave energy than the cylinder because of the amount of flat surface that reflect waves. Inded such reflection may cause interferences that attenuate the wave energy.

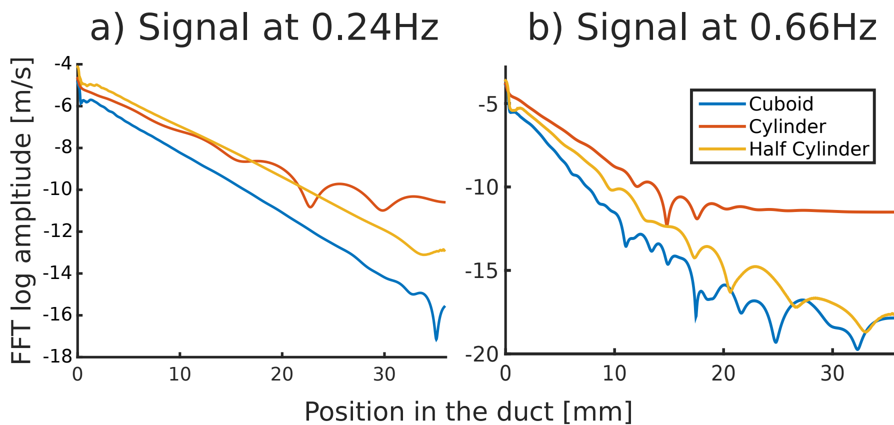


Figure 2.4.: Comparison of single-sided velocity spectrum amplitude at isolated frequencies : 0.24 and 0.66 hertz.

At the imposed 0.24 and 0.66 hertz input frequencies (Fig. 2.4), waves in the cylinder are more stably propagated than in the cuboid. The half cylinder mixes the structural

properties of both other geometries and thus lies in between them in term of wave propagation. While the absorption of wave velocity along the membrane differs for each geometries, it doesn't impact the key phenomena that we are observing : high frequencies waves dissipate faster than low frequencies ones along the duct.

2.1.2. Structural characteristics of the vestibular duct

Cross section of cochlea ducts does not linearly decrease from base to apex. In the human inner ear, the cross section of the vestibular duct varies frequently. Close to its base, a large chamber progressively tighten over approximately 6 millimetres. It then reaches a thin plateau until a bump starts and reaches its end 10 millimetres farther (Fig. 2.5).

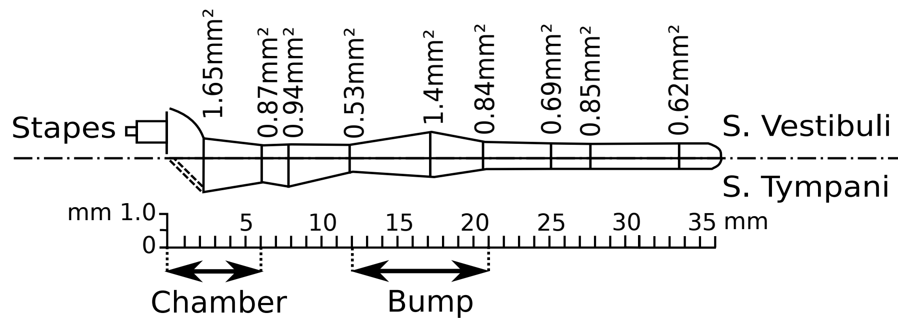
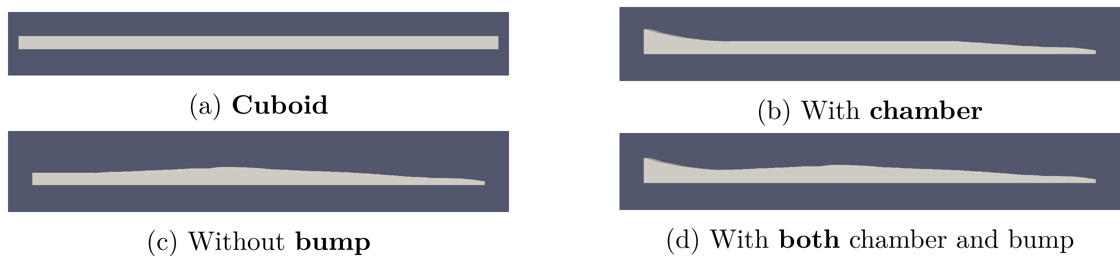


Figure 2.5.: Cross section dimension of the scala vestibuli and main feature of its structure : chamber and bump. Schema and measures adapted from [4, 21, 23].

Such changes of cross section in a waveguide influence the propagated waves differently given their frequency [11]. Indeed, frequencies can be filtered in function of widening or tightening of the the duct. When a duct grow in size over a given distance, it acts as a low-pass acoustic filter. A low-pass filter propagate waves with frequencies lower than its cut-off frequency while attenuating higher frequencies. A high-pass acoustic filter is the opposite, it occurs when the duct shrink over a given distance and passes frequencies higher than its cut-off frequency.



(a) **Cuboid**

(b) With **chamber**

(c) Without **bump**

(d) With **both** chamber and bump

Figure 2.6.: Geometries compared in order to measures the frequency filtering effect of their structure.

The observed characteristics of the vestibular duct let support the presence of such filtering effects. Therefore, we compared four different geometries (Fig. 2.6) in order to monitor if such filtering take place. We used the (a) cuboid geometry as reference having no filtering effects. Its cross section stands constant over all the duct length. The two main features of the vestibular duct are then evaluated separately by modifying the base geometry. The first one (b) is the widening at the base of the duct that form a chamber over 6 millimetres. The second one (c) is the widening in the middle of the duct that form a bump over 10 millimetres. Finally, we combined both feature (d) in order to observe the potential joined filtering effects.

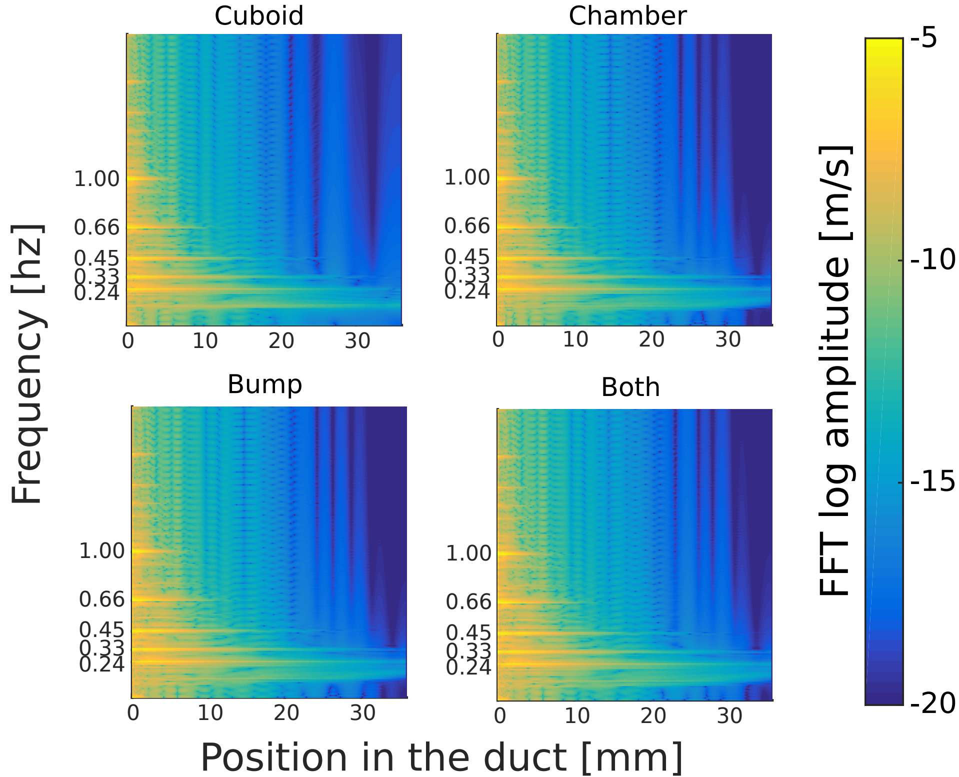


Figure 2.7.: Overview of the single-sided spectrum amplitude of the velocity along the membrane.

The measuring setting was the same as in the previous simulations : the same five frequencies were imposed on the lid of each geometries and the velocity was measured along the duct. We extracted from these measures the velocity single sided spectrum amplitude along the membrane using a discrete Fourier transform. Figure 2.7 gives an overview of the velocity spectrum amplitude against the frequency and duct location. Once again, the five input frequencies are well propagated on each geometry.

Figure 2.8 details more accurately the velocity amplitude of each input frequency in the geometry (c). These measures shows that once again the decay of velocity amplitude is quicker for the *higher* frequencies. In addition, this figure shows that the signal becomes

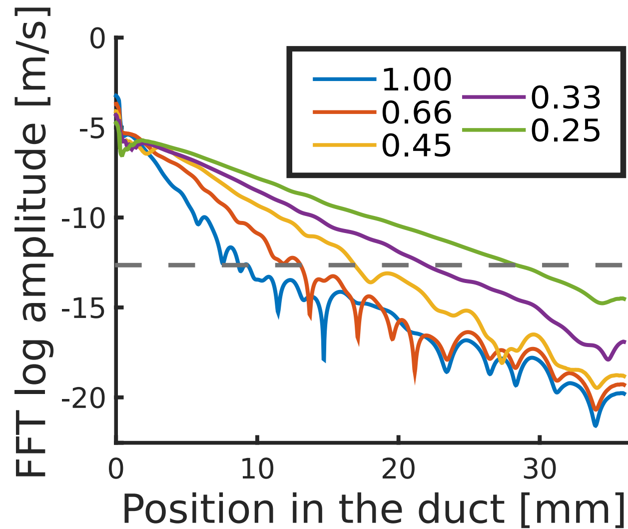


Figure 2.8.: Single-sided spectrum amplitude of the velocity along the membrane at input frequencies for the geometry with both features. The grey dotted line represents the observed noise threshold on high frequencies signal.

noisy and unstable when the velocity amplitude reaches $10^{-12.5} m/s$.

In order to quantify these filtering effects, we compare the velocity amplitude at 0.24 and 0.66 hertz of modified geometries (b,c,d) with the one of the base rectangular geometry (a). The amplifications are showed for amplitude greater than $10^{-12.5}[m/s]$. Under this value noise appears and disrupts the measured velocity amplitude.

Figure. 2.9 reveals that the chamber has a considerable impact on waves at both frequencies. It amplifies them by at least a twofold factor. In between the chamber and the bump, lies a small plateau that seems to act as a high-pass filter. The low frequency signal is slightly attenuated in this region while the high frequency signal remains untouched. The effect of the bump is relatively more difficult to quantify. While it seems that it amplify the low frequency signal, it has a rather unsettling effect on the high frequency signal. Given that in the bump vicinity, the measured velocity amplitude is subject to noise, both effects are hard to assess with confidence.

From these measures we conclude that the structure of the vestibular duct filters wave in function of their frequencies. The chamber seems to amplify signals, more so high frequencies one. The small plateau following seems to maintain the frequencies of high frequencies waves while attenuating lower one. The bump probably amplify wave at low frequencies. Our analyses being based on the velocity norm, we hypothesize that a finer study of each separate velocity components would help to better understand the effect of these duct cavity.

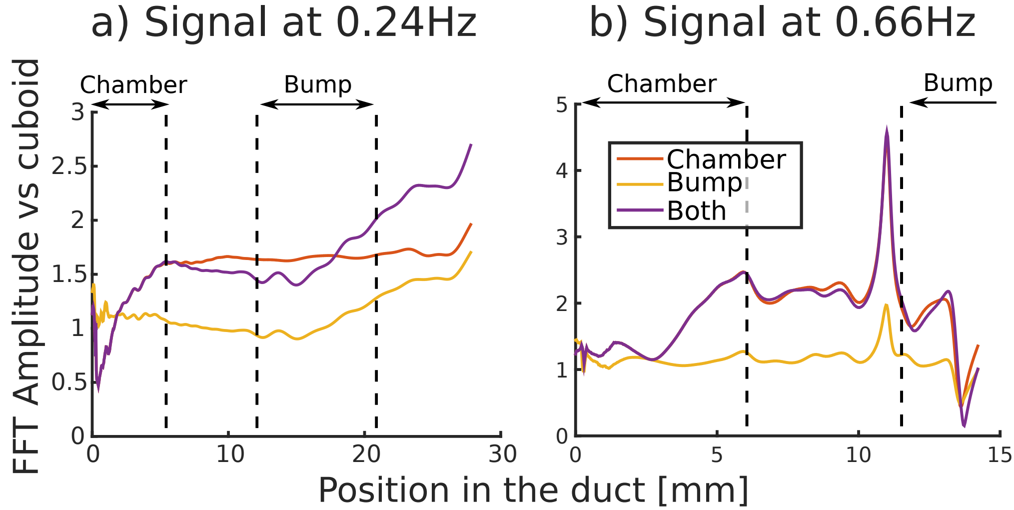


Figure 2.9.: Amplification of the velocity spectrum amplitude of modified geometries when compared to the cuboid.

2.2. Emulating the human cochlea conditions

In our previous simulations, we were focused on analysing the vestibular duct structural effects on the propagated waves. In order to ensure the stability of these simulations, we used physical properties differing strongly from the one of mammals. In this section, we aim at increasing the realism of our simulation by using the physical properties of the human ear. In order to achieve such objectives, two points have to be addressed. In a first time, we have to insure stable simulations based on the properties of the human cochlea. Then, in a second time, we have to directly measure the deformation of the Reissner membrane.

Simulation setting b) : In order to fulfil the first point, we considered the viscosity of the perilymph and the endolymph. These fluids cinematic viscosity is of $8 \cdot 10^{-7} m^2/s$ that correspond to mineralized water [23]. The characteristic dimension of the input membrane was set to 10^{-3} meters with a duct length of $36 \cdot 10^{-3} m$. Lattice units corresponded then to $\delta x = 2 \cdot 10^{-5} m$, $\delta t = 10^{-5} s$ and a Reynolds value of 125. This setting required, per simulation, an average of 20 millions lattice sites that were simulated over 20'000 iterations on 48 processing units for a total simulation time of one day.

In the following experiments, we only considered one frequency at a time. Such choice was made to stabilize the simulation and reduce the total runtime for each of them. Indeed, simulating two frequencies separated by a hundredfold factor, i.e. 50 and 5000 hertz, require a considerable computational effort. This comes from the need to guarantee a sufficient sampling rate for the high frequency wave by decreasing δt and to generate enough iterations to measure multiple period of the low frequency wave. A gross estimation of the time increase gives that covering 50 and 5000 hertz in the same simulation would then take 100 times longer than just one of them.

We arbitrarily chose an input vibration frequency of 100 hertz. This frequency pro-

duced stable simulations and solicited fluid movement in a large portion of the duct. While being less stable, simulations with input vibrations at higher frequencies showed similar phenomenons and maintained the ones observed in figures 2.5 and 2.6 : high frequencies waves dissipated faster than low frequencies ones along the duct.

2.2.1. Attenuation and absorption of waves

Our simulation setting, illustrated by figures 2.1, reveals that the external side of the membrane is in contact with air. However in the inner ear, the cochlear duct sits at the other side of the Reissner membrane. Palabos doesn't yet offer to simulate immersed time-dependent elastic boundary. This is problematic since moving a membrane immersed by salty water on both side does not equate to moving a membrane with air on one side.

In addition to model inaccuracies, we have to deal with multiple other knowledge gaps such as the lack of data on the Reissner membrane mass and elastic properties. Therefore, we postulate that replacing the air present on one side of the membrane by a fluid would absorb kinetic energy and thus slows the membrane displacement. We accordingly tune the shell density and the friction force in order to absorb waves energy and to produce simulation settings closer to the real environment of the human cochlea.

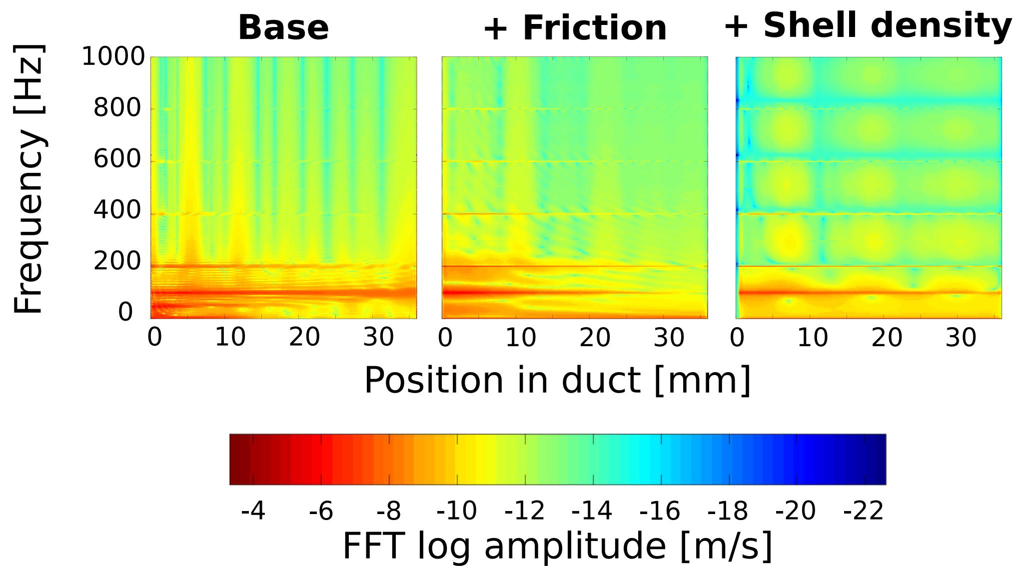


Figure 2.10.: Comparison of the impact of the Reissner membrane shell density and friction force parameters when subject to an 100 hertz input vibration on the oval window. The single-sided velocity spectrum amplitude is showed with, from left to right : 1) The base model without friction and low shell density; 2) Addition of membrane friction; 3) Increase of shell density.

As in previous experiments, we extract the one-sided spectral amplitude of the velocity obtained from the probes vector lying close to the membrane (Fig. 2.10). The

base simulation correspond to the *simulation setting b)* defined previously and an input vibration at 100 hertz. With this setting, the amplitude at 100 hertz is so strong that it goes all the way trough the duct and bounce on the opposite lid. Moreover, the lack of energy absorption creates a strong harmonic wave that appears at 200 hertz. In order, to reduce this phenomena we added a friction coefficient to the membrane. This friction reduced strongly the amplitude of the velocity but amplified the harmonics of the main frequency. Increasing the shell density hindered this effect and produced a clear signal at 100 hertz with weaker harmonics.

2.2.2. Measuring Reissner membrane deformation

While the velocity of the fluid offers a good indication of the Reissner membrane movement, we need a more accurate measure of its deformation in order to link both the vestibular and the cochlear duct. Indeed, our experiment goal is to monitor the effect that a vibration on the lid of the vestibular duct would induce in the microscopic organ of Corti resting in the cochlear duct. Linking accurately both duct is therefore of utmost importance.

In this experiment, we measure the physical displacement of the membrane. For that, we have to remember that this membrane is represented by a set of membrane particles that *freely* evolve in a three dimensional space. More precisely, tenth of thousands of particles are required to correctly represent the membrane. Tracking such amount of particles over tenth of thousand of iterations is a daunting task : it would require a large amount of memory.

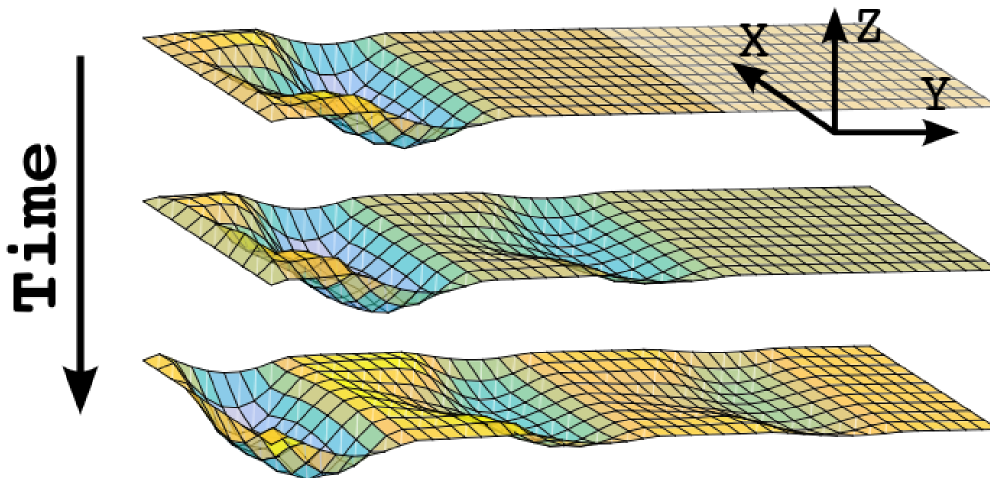
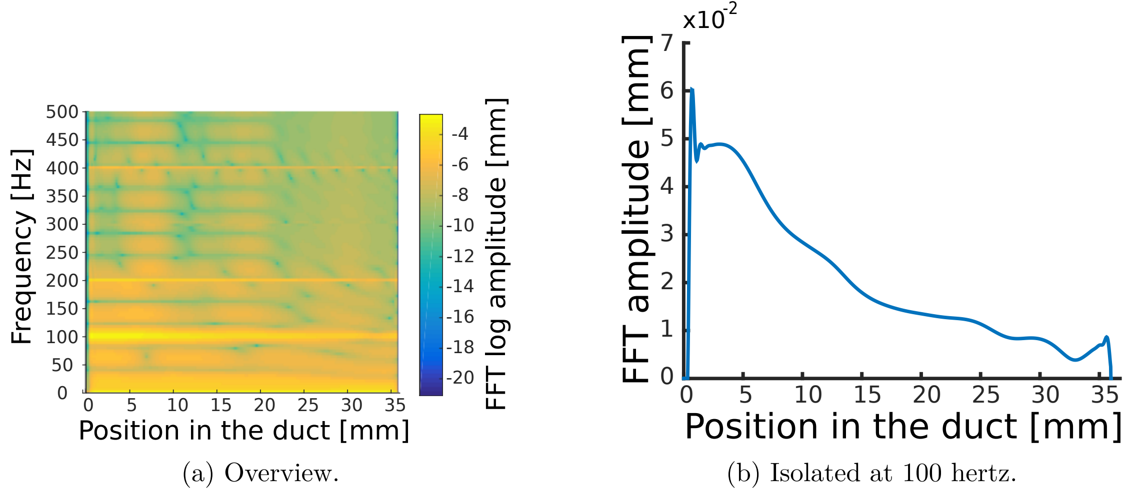


Figure 2.11.: Measured deformation of the membrane at various time steps. Colours represents the height of the membrane going from blue to yellow to red.

In order to reduce the cost of measuring the membrane deformation, we used the two-dimensional (X- and Y-axis) flat plane formed by the membrane at rest as reference. We then reduced the number of points required to memorize the deformation by

using a discrete representation of this plane, with a coarse resolution when compared to the number of membrane particles. Instead of tracking all particles, we measured the membrane deformation only in height (Z-axis) at each fix points (X- and Y-axis) of this discrete plane. Figure 2.11 illustrate deformations measured at different time steps.



(a) Overview.

(b) Isolated at 100 hertz.

Figure 2.12.: Single-sided amplitude spectrum of the membrane deformation.

Using the previously defined realistic conditions, we memorized the membrane deformation in response to an input vibration at 100 hertz. The choice of this frequency was made in order to observe a deformation all along the membrane. We extracted the central vector of deformation measures in respect of the X-axis. We choose this measures vector as it represented the biggest observed membrane deformation over time and analysed its one-sided spectrum amplitude.

Figure 2.12a shows an oscillation of the membrane at 100 hertz with weaker harmonics at 200 and 400 hertz. These observations corroborate the one previously made on the fluid velocity. Figure 2.12a isolates the membrane displacement amplitude for the 100 hertz frequency. The amplitude starts with a value of 50 microns and reduces until the end of the membrane where it reaches a value of 10 microns. The order of magnitude of this simulated deformation is in the same range as the measured thickness of the Reissner membrane (5-15 microns [17]).

It is rather interesting to note that this amplitude begins to drop after the structural chamber of the vestibular duct. In section 2.1.2, we stipulated that the structural plateau in this region may act as a high-pass filter. And indeed, the major part of the deformation attenuation happens in between 6 and $11 - 15$ millimetres which roughly correspond to the end of the chamber and the start of the bump.

3. Simulation of the cochlear duct and the organ of Corti

In the previous chapter, we have shown that periodic waves propagating in the vestibular duct produce oscillation of the Reissner membrane of equal frequencies. This membrane connects the vestibular duct to the cochlear duct. Therefore, this oscillation may well create fluid movements in the cochlear duct and propagate to the microscopic organ of Corti. In this chapter, we enquire if such phenomena could happen and explain how this organ micro-mechanics translates mechanical waves into nervous signals.

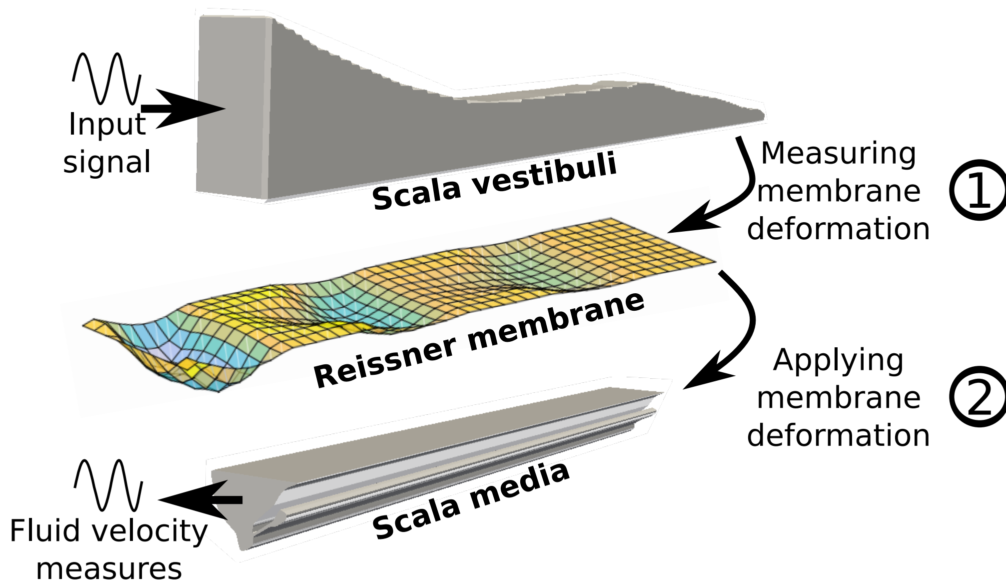


Figure 3.1.: Simulation setting for the scala media. Deformation measure obtained during the first phase are applied to the Reissner membrane. Vectors of probes are placed at key points in the duct in order to measures temporal and spatial fluid velocity.

Figure 3.1 shows the two main steps of this experiment. The first one correspond to the measures of the Reissner membrane deformation with the *simulation setting b)* described in section 2.2. In the second step, we applied this deformation on the the geometry representing the cochlear duct.

However, this geometry, represented in figure 3.2, is far more complex than the one of the vestibular duct and thus was designed more accurately. Such level of detail was required to correctly simulated the fluid velocity inside the organ of Corti. Therefore this

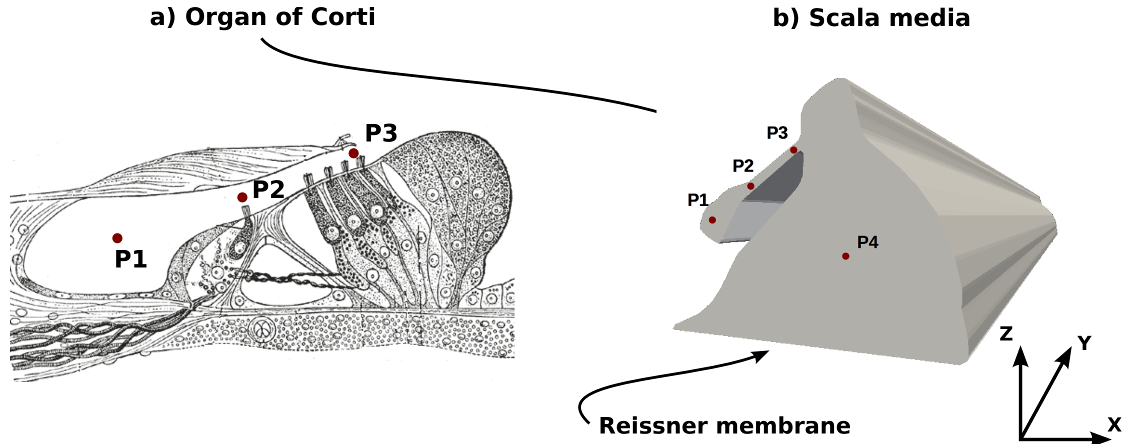


Figure 3.2.: Figure a) shows the biological representation of the organ of Corti and the placement of the probes. Figure b) shows the scala media, horizontally flipped with respect to its representation in figure 3.1, and the probes positions.

canal proportion and global aspect were carefully preserved during its conception. The points of interest lies in the organ of Corti (P1), the scala media (P4) and the tiny canal linking both elements (P2, P3). Fluid velocity of each of these locations will therefore be measured.

3.1. Coupling both simulations

In order to guarantee the coupling of simulation, as shown in figure 3.1, the plane formed by the Reissner membrane was used as a reference. However, while the vestibular duct geometry is long of approximately 35 millimetres, as in reality, we modelled the cochlear duct with a length of one third its true size (10 millimetres). This choice was made to reduce the computing complexity of the simulations.

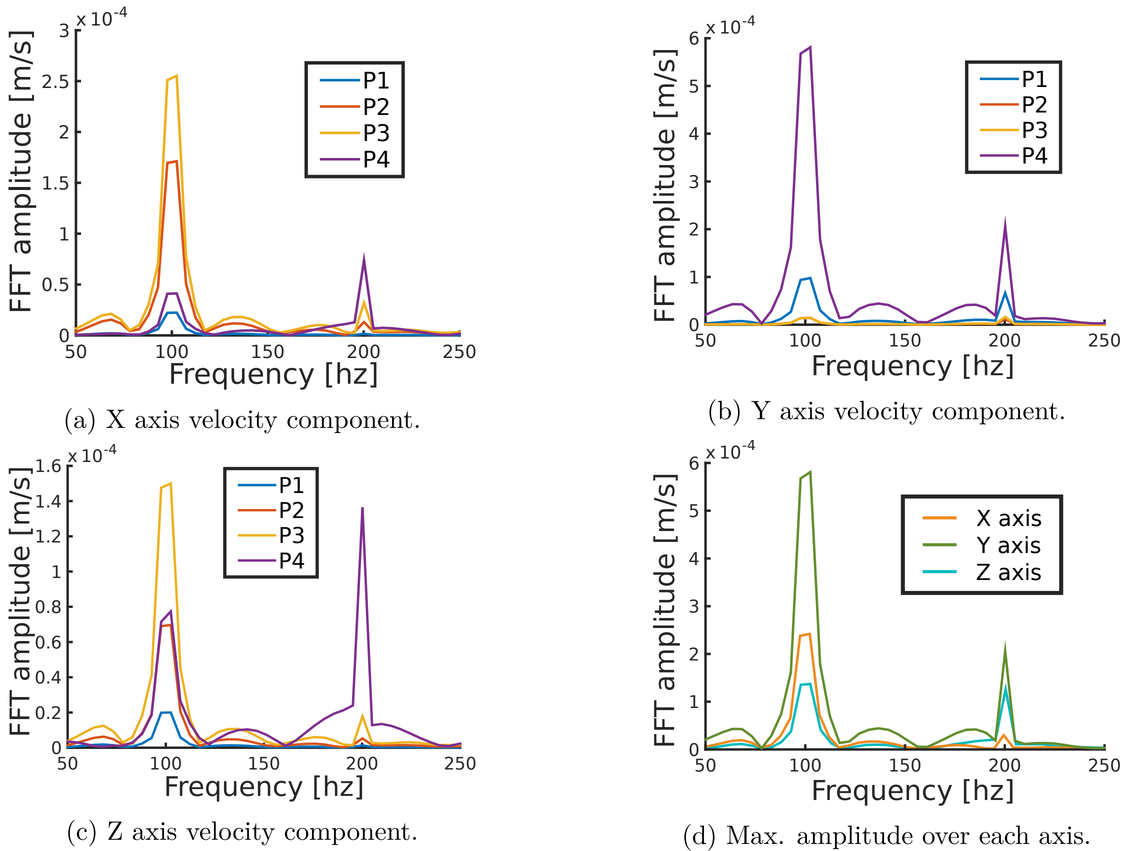
The number of lattice sites in the simulation domain is function of the geometry dimension and the simulation spatial resolution. This resolution is conditioned by the smallest structure in the geometry. In this case, we speak of the entrance of Corti's organ (P2, P3) with a cross sectional size of less than 20 microns. With the duct length of 35 millimetres as the biggest dimension, the size ratio between the tiny canal and the duct length would be greater than 1500. Therefore, we chose a length of 10 millimetres in order to correctly observe the velocity in the first third cochlear duct. Length that already produce a computationally challenging simulation having a size ratio of 500 between its smaller and bigger elements.

Simulation setting c) : The *simulation setting b)* described in section 2.2 were used for this simulation. The spatial resolution of the second simulation was however increased in order to adequately represent the details of Corti's organ ($\delta x = 10^{-6}m$). One third of the cochlear duct required 6 millions of lattice sites to be represented which

were simulated over 20'000 iterations on 96 processing cores for a total simulation time of one day. This duct complex geometry explains the increase of computing power when compared to simulation of the vestibular duct. Indeed, computation of fluid interactions with boundary layers cost more than the computation of inner fluid sites.

3.2. Velocity of the fluid in the scala media

We used four probes in the middle of the duct (middle of Y-axis) represented on figure 3.2 : $P1, P2, P2, P4$. The fluid velocity in the duct was measured at each of those points. For this simulation, the separate velocity components along the X, Y and Z-axis were recorded in order accurately analyse the direction of the fluid. The single-sided velocity amplitude spectrum at each of these point was computed by discrete Fourier transform.



(a) X axis velocity component.

(b) Y axis velocity component.

(c) Z axis velocity component.

(d) Max. amplitude over each axis.

Figure 3.3.: Single-sided spectrum amplitude of the velocity for probes in the middle of the cochlear duct.

These measures (Fig. 3.3) shows that the fluid mainly pulse at the input frequency of 100 hertz with a weak signal at the first harmonic. The strongest fluid flow goes along the Y-axis (Fig. 3.3b) inside the cochlear duct (P4) and at a smaller extend into Corti's organ (P1). This tenuous flux inside Corti's organ is induced by a fluid flow from

the main duct through the canal (P2, P3). Indeed, measures along the X and Z axis (Fig. 3.3a and 3.3c) show a flow inside the canal having the same magnitude as the one inside Corti's organ. The velocity peak along the Z axis inside the cochlear duct is more difficult to explain. We postulate that this flow could be induced by the reflection of the waves formed by the membrane on the top of the cochlear duct. Moreover, multiple features not taken into account during these simulations, such as the cochlea coiling or the softness of the duct wall and tectorial membrane, would certainly hinder reflection of the fluid flow and thus diminish the harmonics velocity amplitude.

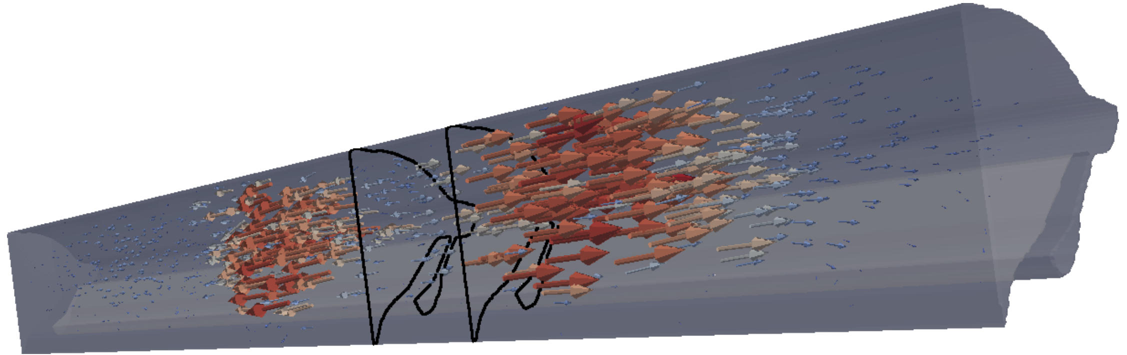


Figure 3.4.: Effect of the membrane deformation, present in between the two black cut, on the fluid velocity. Arrows represents the fluid velocity direction. Their size and color,from small blue to big red, indicate the velocity amplitude.

These observations were confirmed by volume snapshots of the simulation. Such snapshot contains the pressure and velocity components of each particles in the simulated domain at a given iteration. Using the software ParaView[1], these informations can be visualised in three dimension. Figure 3.4 represents the state of the simulation after 15'000 iterations. The arrows in the duct express the fluid velocity with size and color indicating their amplitude (small and blue means slow, big and red means fast). The two black cross-sectional cuts delimit the current wave evolving along the membrane. This wave appears to push the fluid on both its sides, creating two flows of fluid in opposite direction.

Figure 3.5 shows a closer point of view of the current wave emplacement on the membrane. The streamline indicates the path taken by a fluid particle in the duct. Red streamlines indicate higher velocity particles path and are present mainly in the main duct along the Y direction. Blue streamlines show weaker velocity particle paths starting from the wave travelling on the membrane. These streamlines then reaches the opposite side of the main duct (along Z-axis), penetrates through the thin entrance of Corti's organ (along X- and Z-axis) and finally travel along the smaller duct (Y-axis).

These simulations results clearly demonstrated that a fluid flow with low velocity pulsed into Corti's organ in response to an input vibration, more so at the same fre-

[1]ParaView is an open-source, multi-platform data analysis and visualization application. http://www.paraview.org/

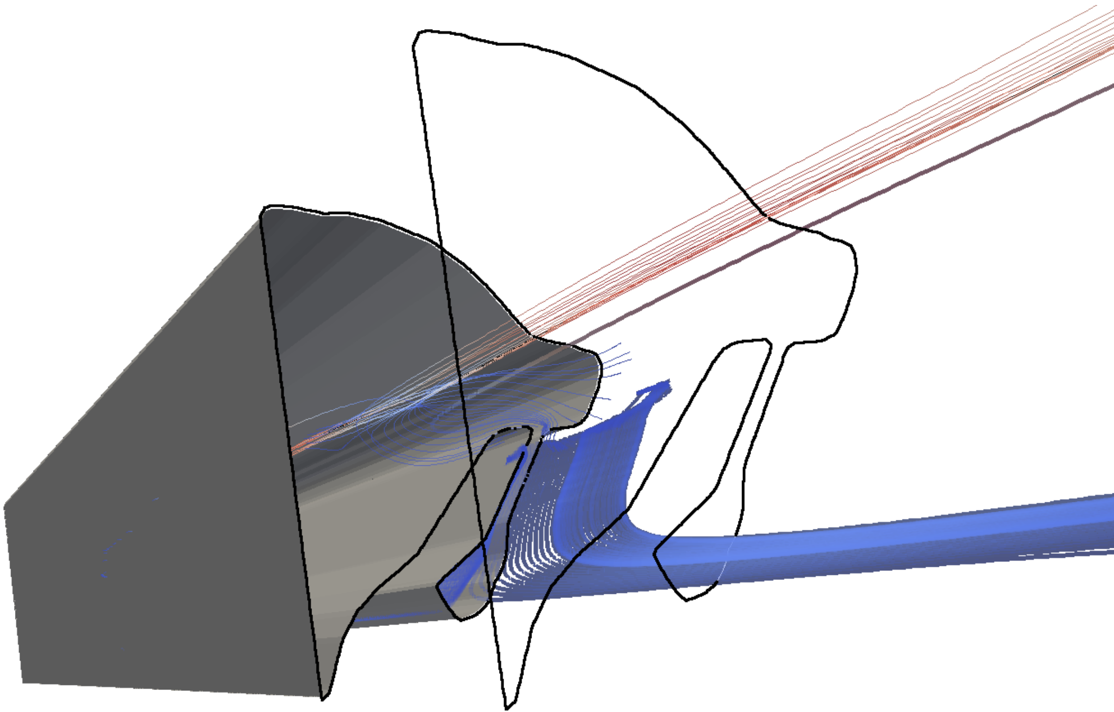


Figure 3.5.: Effect of the membrane deformation, present in between the two black cut, on the fluid path. Streamline represents the path of a particle if it was dropped in the duct. Streamline colors indicate the velocity amplitude of the particles, with blue for slow and red for fast velocity.

quency. As shown in figure 3.1, this input vibration emulated the effect of the ossicles, when excited by sound, on the oval window. Such vibrations then propagated mechanic waves through the perilymph of the vestibular duct inducing an oscillation of the Reissner membrane. This oscillation, preserving the input frequency, created a fluid flow in the cochlear duct that pulsed with low velocity at the exact position of the hair cells.

In this scenario, the Reissner membrane plays a central role by relaying the initial vibrations imposed to the oval window into the cochlear duct. This, by itself, contravene the implicitly and widely accepted scenario in which the Reissner membrane doesn't affect the fluid flow between the vestibular and cochlear duct. Indeed, removing this elastic membrane would strongly impact the dynamic of the fluid flow.

4. Discussion and Conclusion

4.1. Inner ear mechanism

One of the inner ear function is to transform the sound that surrounds us into nervous signals. This mechanism is decomposed in multiple steps starting from the vibration of the tympanic membrane. These vibrations are amplified by the ossicles and transmitted to the liquid filled vestibule through the oval windows. The induced fluid waves generate a vibratory response in the cochlea. This vibratory response is attributed to the difference of pressure in the oval and round windows that would create a travelling wave on the basilar membrane. This travelling wave activate then hair cells, or stereocila, present in Corti's organ due to its closeness with the basilar membrane [23].

This simplistic scenario minimizes the potential function of the undervalued Reissner membrane [14]. Recently, this membrane has been shown to propagate travelling waves with amplitude comparable to the one of the basilar membrane [15]. We simulated this phenomena by applying vibrations at the entrance of the vestibular duct. These vibrations propagated waves inside the duct, thus deforming the Reissner membrane. The oscillation of the membrane maintained the input vibrations frequency and further induced fluid flows in the cochlear duct. Most of the fluid movement went along the membrane, but more importantly, a fluid flow, having half the velocity of the main one, went inside the microscopic organ of Corti.

While being weaker, this flow pulsed with the same frequency of the input vibrations at the exact location of the sterocilia. These hair cells, depending on their sensibility, could thus have been activated and emitted nervous signal to the brain. Therefore, these results corroborate our initial hypothesis, that movement of the Reissner membrane could be of utmost importance in the mechanism of hearing and yet remains nearly unconsidered nowadays.

In light of the previous results, the ducts structures may play a pivotal role by filtering the perceived sound frequencies. Filtering effects are known to affect waves travelling inside a waveguide of varying cross sectional size [11]. Cross section measured from the base to the apex of the scala vestibuli and tympani [4, 21, 23] variates multiple times and form various chambers and bumps. We compared a variety of geometries representing part of these structural features and the ones respecting the proportions of the vestibular duct amplified or attenuated the fluid velocity by more than a twofold factor. These variation in fluid velocity clearly coincided with the structural characteristics of the duct. The variety of structural features of the vestibular and tympanic duct in various mammals [22] could well impact the range of sound perceived by each species.

4.2. Simulations limitations and challenges

While striving to emulate the human ear conditions, our simulations remained far from the reality. While we linked the vestibular and cochlear duct using the Reissner membrane, during the separate simulation steps, the membrane was surrounded on one side by a fluid mimicking the perilymph and by air on the other side. The impact of this inaccurate setting was minimized by increasing the density and friction of the membrane. However, these required simplifications created a gap between our model and the real environment of the cochlea.

Accurately simulating the cochlea poses serious challenges for multiple reasons. One of them resides in the complex structure and anatomy of the inner ear. Despite the recent progress in high-resolution medical imaging, soft tissues such as the Reissner membrane remains hard to observe and measure accurately [4, 17]. Assuming that these issues will be fixed in the future, fully accurate geometries will then cause considerable computational challenges for CFD solvers.

The frequencies range and the scale of geometry details featured in the cochlea leads to hardly tractable computational task. Simulating low and high frequency at the same time requires to chose a time resolution that enable the sampling of the fastest frequency and a run time long enough for the lowest frequency to propagate. Sampling 20 points per period of a 20'000 hertz signal requires a time step of $5 \cdot 10^{-6}$ seconds. With such time step, 10^4 iterations would be necessary to just generate one period of a 20 hertz signal. The same observation applies to the spatial resolution of the geometry. As example, Corti's organ has geometric features of less than 10 microns, while the unrolled length of the cochlea is greater than 30 millimetres.

The spatial and temporal resolutions of our simulation of one third of the cochlear duct already required a one day runtime with 96 processors to generate enough data. This small simulation, in regard to the entire cochlea, was not even computing forces on the membrane since we applied its previously memorized deformation. Yet tenth of thousands of particles were used to represent accurately the membrane. Adding the resolution of forces on the entire Reissner and basilar membrane would consequently increase the required computational power.

Hopefully, advances in CFD will help address these issues. Multi-domains methods [12, 19] are a first example of such methods. They aim to facilitate the simulation of geometries showing structure with multiple scales of dimensions by dividing the simulation domain in blocks having each an appropriate spatial resolution. Another issue comes from immersed elastic membrane. Such complex fluid-solid interactions are complex to address with the Lattice-Boltzmann method and are subject to recent advances [6].

In addition to these CFD improvements, a direct solution to this challenging computational task would be to use the already available computing resources. Palabos is known to manage simulation on clusters having several thousands of processors[1].

[1]LEMANICUS Blue Gene/Q Supercomputer : http://bluegene.epfl.ch/

4.3. Conclusion

In conclusion, by simulating the full path of mechanical waves created by vibrations of the oval window, we showed that the Reissner membrane could well play an important role in the activation of hair cells present in the microscopic organ of Corti. In this context, the structure of cochlea ducts presented significant frequency filtering properties by modifying wave velocity by up to a twofold factor.

In order to simulate these phenomenon, we used a highly parallel CFD solver, Palabos. While these experiments where not fully realistic in regard of the human cochlea, they present an important step toward the simulation of inner ear micro- and macro-mechanics. Novel CFD methods coupled with the increasingly accurate measures of the human ear will allow in a near future to conceive truly realistic cochlea models. Given that the required computing resources are already us available for such computational challenge, we only are a few steps away of realising the crucial tool that would help us better understand the human hearing mechanism.

Bibliography

[1] Békésy, Georg V. “The Variation of Phase Along the Basilar Membrane with Sinusoidal Vibrations.” The Journal of the Acoustical Society of America 19, no. 3 (May 1, 1947): 452–60. doi:10.1121/1.1916502.

[2] Békésy, Georg von, and Ernest Glen Wever. Experiments in Hearing. New York: McGraw-Hill, 1960.

[3] Böhnke, F., and W. Arnold. “Nonlinear Mechanics of the Organ of Corti Caused by Deiters Cells.” IEEE Transactions on Bio-Medical Engineering 45, no. 10 (October 1998): 1227–33.

[4] Braun, K., F. Böhnke, and T. Stark. “Three-Dimensional Representation of the Human Cochlea Using Micro-Computed Tomography Data: Presenting an Anatomical Model for Further Numerical Calculations.” Acta Otolaryngol 132, no. 6 (2012): 603–13.

[5] Elliott, Stephen J., and Christopher A. Shera. “The Cochlea as a Smart Structure.” Smart Materials & Structures 21, no. 6 (June 2012): 64001.

[6] Favier, Julien, Alistair Revell, and Alfredo Pinelli. “A Lattice Boltzmann–Immersed Boundary Method to Simulate the Fluid Interaction with Moving and Slender Flexible Objects.” Journal of Computational Physics 261 (March 15, 2014): 145–61.

[7] Geisler, C. Daniel, and Chunning Sang. “A Cochlear Model Using Feed-Forward Outer-Hair-Cell Forces.” Hearing Research 86, no. 1–2 (June 1995): 132–46.

[8] Givelberg, Edward, and Julian Bunn. “A Comprehensive Three-Dimensional Model of the Cochlea.” Journal of Computational Physics 191, no. 2 (2003): 377–91.

[9] Huber, Christian, Babak Shafei, and Andrea Parmigiani. “A New Pore-Scale Model for Linear and Non-Linear Heterogeneous Dissolution and Precipitation.” Geochimica et Cosmochimica Acta 124 (January 1, 2014): 109–30.

[10] Kemp, David T. “Otoacoustic Emissions, Their Origin in Cochlear Function, and Use.” British Medical Bulletin 63, no. 1 (October 1, 2002): 223–41.

[11] Kinsler, Lawrence E, Austin R Frey, Alan B Coppens, and James V Sanders. “Fundamentals of Acoustics.” Fundamentals of Acoustics, 4th Edition, by Lawrence E. Kinsler, Austin R. Frey, Alan B. Coppens, James V. Sanders, Pp. 560. ISBN 0-471-84789-5. Wiley-VCH, December 1999. 1 (1999).

[12] Lagrava, D., O. Malaspinas, J. Latt, and B. Chopard. “Advances in Multi-Domain Lattice Boltzmann Grid Refinement.” Journal of Computational Physics 231, no. 14 (May 20, 2012): 4808–22.

[13] Manoussaki, D., and R. Chadwick. “Effects of Geometry on Fluid Loading in a Coiled Cochlea.” SIAM Journal on Applied Mathematics 61, no. 2 (January 1, 2000): 369–86.

[14] Ni, Guangjian, Stephen J. Elliott, Mohammad Ayat, and Paul D. Teal. “Modelling Cochlear Mechanics.” BioMed Research International 2014 (July 23, 2014): e150637.

[15] Reichenbach, Tobias, Aleksandra Stefanovic, Fumiaki Nin, and A. J. Hudspeth. “Waves on Reissner’s Membrane: A Mechanism for the Propagation of Otoacoustic Emissions from the Cochlea.” Cell Rep 1, no. 4 (April 2012): 374–84.

[16] Robles, Luis, and Mario A. Ruggero. “Mechanics of the Mammalian Cochlea.” Physiological Reviews 81, no. 3 (July 2001): 1305–52.

[17] Shibata, Takashi, Sumiko Matsumoto, Tetsuzo Agishi, and Teiko Nagano. “Visualization of Reissner Membrane and the Spiral Ganglion in Human Fetal Cochlea by Micro-Computed Tomography.” American Journal of Otolaryngology 30, no. 2 (2009): 112–20.

[18] Steele, Charles R., Jacques Boutet de Monvel, and Sunil Puria. “A MULTISCALE MODEL OF THE ORGAN OF CORTI.” Journal of Mechanics of Materials and Structures 4, no. 4 (2009): 755–78.

[19] Touil, Hatem, Denis Ricot, and Emmanuel Lévêque. “Direct and Large-Eddy Simulation of Turbulent Flows on Composite Multi-Resolution Grids by the Lattice Boltzmann Method.” Journal of Computational Physics 256 (January 1, 2014): 220–33.

[20] Ulfendahl, Mats. “Mechanical Responses of the Mammalian Cochlea.” Progress in Neurobiology 53, no. 3 (October 1997): 331–80.

[21] Wysocki, Jaros aw. “Dimensions of the Human Vestibular and Tympanic Scalae.” Hearing Research 135, no. 1–2 (1999): 39–46.

[22] Wysocki, Jaros aw. “Dimensions of the Vestibular and Tympanic Scalae of the Cochlea in Selected Mammals.” Hearing Research 161, no. 1 (2001): 1–9.

[23] Yost, William A, and DW Nielsen. ”Fundamentals of Hearing”. Holt, Rinehart, and, 1989.

[24] Zeng, Fan-Gang, and Richard R. Fay. Cochlear Implants: Auditory Prostheses and Electric Hearing. Springer Science & Business Media, 2013.

[25] Zimny, Simon, Bastien Chopard, Orestis Malaspinas, Eric Lorenz, Kartik Jain, Sabine Roller, and Jörg Bernsdorf. "A Multiscale Approach for the Coupled Simulation of Blood Flow and Thrombus Formation in Intracranial Aneurysms." Procedia Computer Science, 2013 International Conference on Computational Science, 18 (2013): 1006–15.

[26] Zwislocki, J. J. "Cochlear Waves: Interaction between Theory and Experiments." The Journal of the Acoustical Society of America 55, no. 3 (March 1, 1974): 578–83.

Part III.

Toward a high performance framework for statistical inference in evolutionary biology

Preamble

After being conceived in the mid-19th century on the HMS Beagle, Darwin's theory of evolution matured over the years until a game changing event occured: the advent of DNA sequencing. From then on, statistical models capturing the essence of this theory have been thoroughly scrutinised in light of this novel abundance of evolutionary evidence. Thanks to sophisticated statistical methods and the rise of scientific computing, this practice flourished and gave birth some decades ago to the fields of bioinformatics and computational biology [54].

Research efforts from these new fields helped the study of key mechanisms of Darwin's theory of evolution such as the concept of positive selection that explains how advantageous genetic variants in individuals are fixed in the relevant population. In 1994, Muse and Gaut [88] provided a methodology, with unprecedented power, for the detection of positive selection using molecular data. While remaining one of the most popular approaches nowadays, this method revealed to have an overly expensive computational cost when applied on such large datasets.

In addition of impeding the analysis of large molecular datasets, this limitation had the secondary effect to put a hold on the development of models detecting positive selection. Indeed, more realistic and complex models would inherit and most probably increase this already limiting computational cost. Therefore, investigations were launched to identify more efficient algorithms and statistical methods that could benefit from nowadays availability of parallel computing resources.

In this thesis part, we join our effort to this ongoing challenge. However, instead of focusing on the sole problematic caused by positive selection models, we dedicate our effort to the ambitious project of considering this challenge on a larger spectrum: the vast majority of today evolutionary biology models, as well as the ones of tomorrow. Therefore, this study aims at answering the general question:

- Can a high performance and parallel framework for statistical inference of evolutionary models be designed such as to enable the analysis of large datasets with complex and realistic models ?

This part presents our attempt at providing such a framework and is articulated as follows. The first chapter starts by putting the stated challenge within the state-of-the-art and then is dedicated to the introduction of an excerpt of evolutionary models. The next chapter describes a general formulation enabling efficient evaluations of statistical models. Based on this formulation, chapters three and four propose novel algorithms and parallel strategies for two different statistical methods: the maximum likelihood estimation and the Markov chain Monte Carlo methods. The fifth chapter presents a strategy enabling large scale statistical analyses to be conducted on high-performance

computing infrastructure. The methods presented in chapters two to five are illustrated all along this part with experiments on nowadays evolutionary biology challenges and compared to state-of-the-art software. The final chapter synthesises all these approaches such as to justify their usefulness in the context of a high performance framework for statistical inference and provide an outlook of its potential in the present and future of computational evolutionary biology.

1. Introduction

1.1. State-of-the-art

The use of computer science in biology has surged during the last decades and has lead to the establishment of, once controversial, new research domains : bioinformatic and computational biology [92]. These nowadays well-grounded fields are more than ever solicited in light of the huge amount of biological data being generated every days. For example, the generation rate and amount of genomic data competes directly with the one of astrophysics and even the one of internet mastodons such as YouTube or Twitter [122]. Biology entrance in the Big Data era [82] opened new world of perspectives to validate long dating postulates and to investigate novel hypotheses.

Evolutionary biology, the study of evolutionary processes that produced the diversity of life on Earth, has been positively impacted by this abundance of data. Concepts exposed in 1859 by Charles Darwin in his famous book, *On the origin of species*, are nowadays extensively studied using genetic materials, fossil records and biological observations. For example, natural selection, the process favouring the survival and reproduction of organisms that are best adapted to their environment, is nowadays routinely investigated using genomic data [130]. As a second example, the quest for the *Tree of Life*, the evolutionary relationships of all known species through time, seems closer than ever [58].

This richness in data allied to the availability of computational resources made the aforementioned concepts and a vast amount of other hypotheses more approachable. Indeed, after being carefully and realistically expressed as statistical models, these hypotheses are then confronted and compared on datasets using statistical inference. However, the quest of ever more realistic models coupled with datasets of increasing size reached a limiting factor : its computational cost. Indeed, such complex analysis could well take months or years to be achieved.

Numerous approaches are currently used to overcome, at some extent, this limitation. These approaches can be decomposed in two classes : *model-specific* ones and *model-generic* ones. The first class aims to reduce the computational cost, or computational time, of likelihood evaluations of a given statistical model. These model-specific improvements are achieved by using better algorithms, more adapted programming languages and parallel computations when possible.

Detecting positive selection, a mode of natural selection, on coding genes gives a good example of such model evolutions and implementation improvements. The initial models were conceived in the early 1990 [44, 88] and were analysed using frequentist statistics and authors of the models proposed a C implementation based on maximum likelihood (ML) methods : `PAML` [135]. These implementations and models evolved through the

years [136, 137]. In 2002 a more complex and realistic branch-site model detecting positive selection at individual sites along specific lineages was proposed [138]. The fourth version of PAML with new models implementation and algorithms enhancement followed in 2007 [140]. However large analyses on the branch-site model using this implementation were still challenging. For that matter, a faster implementation based on improved and parallel algorithms, for this specific branch-site model, was developed and released in 2014 [108, 127].

Another representative example is the study of evolutionary relationships between species through time, more commonly called phylogenetic inference [33]. The challenging field of computational phylogenetics owns a large variety of implementations. ML based implementations include PhyML [50] offering a rich variety of models, RaXML [120] designed for performance and massively parallel executions, and more. Bayesian approaches, while being usually slower, are nonetheless not fewer. The most notable among these implementations based on Markov Chain Monte Carlo (MCMC) methods are MrBayes [106] a model-rich and optimized framework, Beast [31] a slower version axed toward tree dating and ExaBayes [1] designed for performance and massively parallel executions. Concurrent efforts were made to produce libraries for parallel and highly optimized likelihood computations for key models without considering the statistical framework. Among them Beagle [10] offers parallel likelihood computation on CPU or GPU and the Phylogenetic Likelihood Library (PLL [24]) offers a parallel implementation using optimized SSE3/AVX vectorized instructions.

While being efficient, all these state-of-the-art implementations remain limited by the expensive computational cost encountered when analysing large datasets having either long molecular sequences or representing a large amount of species. To tackle these limitations, software having parallel implementations take advantage of long DNA sequences by sending subsets of sequence positions to various processors in order to distribute the computational cost. However when faced with datasets having large amount of species, this approach does not bring any improvements. In addition of performance limitation, these model-specific implementations represent a huge investment in development and maintenance time (several years for each of them). Investments that could well be lost in case of a shift toward drastically different statistical models for the same applications. Finally, minimal, if no, improvements coming from computational statistics (for Bayesian inference) or optimization methods (for ML) are being employed in those implementations.

Conversely, as to minimise the dependency on models, model-generic approaches offer a flexible framework for their development and analysis. In computational evolutionary biology, RevBayes [59] is a recent initiative that focuses on offering a user friendly environment for biologist to develop their own models that can then be analysed using basic MCMC implementation. However model-generic approaches are more common in computational statistics with software such as *Bayesian inference Using Gibbs Sampling* (BUGS [80]) and *Just Another Gibbs Sampler* (JAGS [96]). Both software offer a larger flexibility and variety in terms of samplers. Another promising initiative Stan [121] offers a user-friendly and flexible environment for model development and use a state-of-the-art Hamiltonian Monte Carlo sampler (HMC [60]) to offer an enhanced sampling

performance.

Apart for RevBayes these generic approaches do not offer any support for evolutionary biology models which is of particular importance given that phylogenetic inference require the sampling of the discrete space representing all the potential tree topologies. Sampling such space represents a major challenge that might prove incompatible with general statistical implementations. For example, the choice of Hamiltonian Monte Carlo as Stan core sampler induces that all model parameters must be continuous. Furthermore these general frameworks lack the algorithmic improvements built in model-specific implementations and more importantly might not be flexible enough to integrate them. In addition of these limitations, none of these frameworks have been designed to take profit of high-performance computing (HPC) resources; feature that becomes nearly unavoidable when analysing complex models on large datasets.

While having each strong points, neither model-specific nor model-generic existing implementations encompass all of them. Therefore, computational evolutionary biology, and probably other research fields, would greatly benefit from a framework for statistical inference offering the following key features:

- **Flexibility and Modularity** such as to enable an evolving and lasting framework that could be enriched by novel statistical models and methods.

- **Efficiency** to allow the analysis of complex models by offering generic parallel algorithms, model-generic improvements and state-of-the-art statistical strategies, while still permitting model-specific improvements.

- **HPC support** by managing multiple levels of parallelism. Model-level parallelism to speed up likelihood computations and enable Big Data applications, with the definition of application having to much data too fit the memory of a single processing unit. Method-level parallelism to speed up and improve the accuracy of statistical methods. Analysis-level to support large statistical analyses requiring several independent computations.

- **User-friendliness** in regards of end-users and developers. End-users require a simplified approach to interact with the available tools, while statistical models or methods developers require clear directives, or interfaces, that their implementations must fulfil.

We present in this thesis part a High perfOrmance framework for Generic stAtistical modelliNg, HOGAN, allying modularity, efficiency, HPC-readiness and user-friendliness. Reaching such goal required techniques from multiple areas of computer science as well as from computational statistics and computational biology. In the spirit of orienting the reader through this thesis part, the next section offers an overview of the content of the following chapters as well as their relevance in regard of the overall framework. The final section of this introduction briefly introduces the applications, or statistical models, that are used to illustrate the framework performance and versatility in the following chapters.

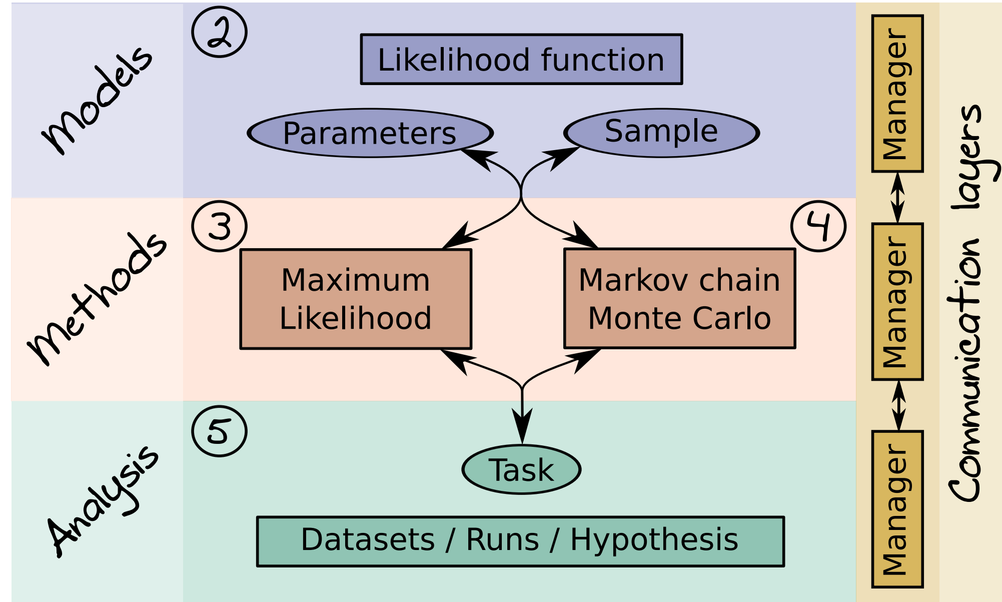


Figure 1.1.: Overview of HOGAN.

1.2. Overview

As illustrated in figure 1.1, HOGAN is mainly composed of three different layers: models, methods and analyses. Each of these three layers disposes of a dedicated MPI communication manager. This approach enables the combined use of parallelism at each layer to cumulate their speedup and thus increase the scaling of the framework.

1.2.1. Models

The first layer is dedicated to the generalization of statistical models and more precisely to the implementation and computation of likelihood functions. This generalization is obtained by expressing the computational operations of the likelihood function as a *Directed Acyclic Graph* (DAG). The second chapter of this thesis part formalizes this approach and describes how it is used to improve the efficiency of HOGAN. The DAG representation enables the generalization of model-specific algorithmic improvements and the use of well known parallel computation strategies.

1.2.2. Methods

The second layer is dedicated to statistical methods and has support for frequentist inference based on ML Estimation (MLE) and Bayesian inference using MCMC methods.

In computational evolutionary biology, ML estimations are frequently achieved using hill-climbing optimization strategies for the likelihood function. For that matter, Newton or Quasi-Newton methods such as the *Broyden–Fletcher–Goldfarb–Shanno* algorithm (BFGS [91]) are employed. Such methods require the approximation of the

finite-difference derivatives of the likelihood function. The third chapter describe how such approximations can be obtained efficiently by using the DAG representation as well as by using parallel computations.

MCMC methods are used in Bayesian inference to approximate the posterior probability distribution of the model parameters given the data. However, to achieve accurate approximation of these posterior distributions, MCMC methods require more likelihood evaluations than their ML counterparts making them slower. In addition to that, they are inherently sequential and require prior knowledge on the likelihood function to define adequate moves in the parameter space.

This limitations are addressed in chapter four by presenting a novel sampling scheme coupling two state-of-the-art computational statistics methods: prefetching and adaptive MCMC. This sampling scheme enables the use of parallel computing resources to speed up the sampling. Furthermore moves adapted to the parameter space are automatically learned and their computation scheme is optimised using the DAG representation of the likelihoods. Finally, this novel model-generic approach, being an enhancement to the base MCMC process, can be coupled with other well known methods such as the parallel tempering [41].

1.2.3. Large scale analyses

Large scale analyses are managed by the third layer and represent the scenarios where multiple tasks are required to achieve an analysis. This vast definition encompasses the analysis of multiple hypotheses that are to be compared within a statistical framework, or replicate analysis required to insure correctness of the results obtained though the statistical methods, as well as applications of a statistical analysis on a large amount of datasets. While such tasks can be independently computed, their execution time remains unknown before hand. The fifth chapter describes the asynchronous distributed load balancing algorithm put in place to manage such situation and facilitate the use of HPC resources

1.2.4. Overall framework

The overall framework is shortly detailed in the sixth and final chapter. This summary of the current version of `HOGAN` is followed by a discussion of the potential enhancements that could be integrated in the framework as well as the future target application that could benefit from it.

1.3. Illustrative applications

The framework capabilities and performance are illustrated in the following chapters using various applications that are summarized in this section. The first one is a toy application based on multivariate normal distributions that will be used to analyse our MCMC sampler properties, while the second one is a hierarchical Bayesian model, `PyRate`, used to illustrate its performance.

The remaining applications are based on phylogenetic trees and molecular evolution models. Codon models identifying positive selection are used to illustrate the DAG representation of likelihood as well as the improvements in ML methods. Basic codon models on fixed trees, as well as the inference of phylogenetic trees using nucleotide models are used to illustrate the MCMC sampler performance. Finally, a nucleotide model estimating the co-evolution of two positions in an alignment, `Coev`, is subject to a large scale analysis.

1.3.1. Multivariate normal based models

Multivariate normal based models are really convenient to analyse the properties of a MCMC sampler on independent and correlated variables. Their likelihood function is defined as the probability of the data X formed by m samples generated from a multivariate normal distribution $\mathcal{N}(\mu, \Sigma)$ with $\mu \in \mathbb{R}^d$ and $\Sigma \in \mathbb{R}^{d \times d}$. Setting $\theta = (\mu, \Sigma)$, the likelihood function becomes $f(X|\theta) = \prod_{k=1}^{m} f_\theta(x_k)$ with f_θ being the probability density function of the multivariate normal distribution given the parameters θ.

Models with d independent parameters are defined by a diagonal covariance matrix $\Sigma = diag(V)$ with $V = (\sigma_1^2, .., \sigma_d^2)$. Models with d correlated parameters are, for their part, expressed with covariance matrices Σ having correlations between several of the d dimensions of the distribution.

1.3.2. PyRate

The model `PyRate` [114] is a hierarchical Bayesian model that analyses speciation and extinction rates of large collections of fossils and estimates large numbers of parameters N. This model has a complexity order of $\mathcal{O}(N)$ with few parameters being correlated. These properties make this model particularly interesting. Indeed the relatively inexpensive likelihood can highlight the overhead of the presented parallel MCMC sampler, while the low amount of correlations can illustrate its ability to exploit any existing levels of correlations.

Another advantage of this model is the fact that data sets can be simulated using the simulator of the `PyRate` model [115]. Moreover, large empirical datasets that are challenging to sample are also available, such as a large dataset of plant fossils [116] containing 22,415 fossil occurrences assigned to 443 plant genera. This dataset spans over a hundred millions of years divided in 31 predefined epochs, which are defined by the stratigraphic geological time scale.

1.3.3. Tree-based models of molecular evolution

Multiple models based on phylogenetic trees and models of molecular evolutions are used in the following chapters to illustrate key features of `HOGAN`. Therefore a brief overview of this class of models is introduced prior to a more detailed description of their respective specificities. This introduction is largely based on the stellar books *Computational Molecular Evolution* [139] and *Inferring Phylogenies* [33].

General description

The genetic material of a species, once extracted, is represented as molecular data that contains symbols identifying DNA, RNA, codon or amino acid sequences. These symbols, or states, forming such sequences are known to evolve by changing from one to another. This substitution process is the key phenomena modelled in molecular evolutionary models.

Such models are using continuous-time Markov chains to define the probability $p_{ij}(t)$ of state i to evolve into state j during time t. This matrix of transition probabilities P can be defined as

$$P(t) = e^{Qt}, \tag{1.1}$$

with Q defining the infinitesimal generator. In molecular evolutionary biology, t is used to represent the expected number of substitutions, while the Q matrix, more commonly called the instantaneous substitution-rate matrix, is used to model evolutionary hypotheses.

The simplest example of such hypothesis is the Jukes-Cantor model of nucleotide substitutions [124]. This model assumes a unique substitution-rate λ between nucleotides. The matrix Q of this hypothesis would be represented as

$$Q = \{q_{ij}\} = \begin{bmatrix} \cdot & \lambda & \lambda & \lambda \\ \lambda & \cdot & \lambda & \lambda \\ \lambda & \lambda & \cdot & \lambda \\ \lambda & \lambda & \lambda & \cdot \end{bmatrix} \tag{1.2}$$

with diagonal values constrained to -3λ such that the sum of each row equals to zero.

These evolutionary models are commonly used to define the evolutionary relationships between species of a phylogeny, or phylogenetic trees, and therefore play a key role in building the likelihood function used in phylogenetic analysis. Such analyses serve two distinct purposes. The first one is to test hypotheses regarding the evolutionary model or to study its parameters θ using a fixed tree topology τ. Oppositely, for their second purpose the tree topology τ is the key parameter that has to be estimated and therefore is part of θ.

In both scenarios, the likelihood represents the probability of observing the molecular data X when the parameters θ are given. The data represent S aligned homologous sequences having each N positions, or sites (nucleotides, codons or amino acids). The kth symbol of the sequence representing the hth species is therefore denoted as x_{kh}. The possible states, or symbols, that x_{kh} can take are defined over the set $\mathcal{K}$ that, in the case of nucleotides, would be defined as $\mathcal{K} = \{A, C, T, G\}$. The parameters θ, for their part, encompass the parameters of the evolutionary model as well as the evolutionary distance, or expected number of substitutions t, between two species and potentially the tree topology τ.

Two important assumptions are made on the evolutionary process to simplify the definition of the likelihood. The first one is that different positions in a sequence evolve independently of each other. This assumption enables the decomposition of the likelihood

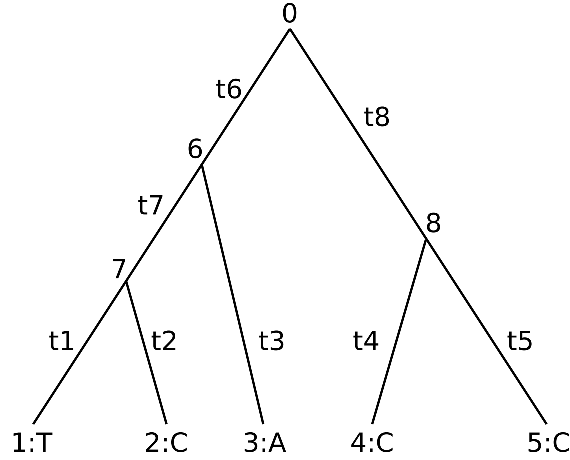


Figure 1.2.: Example of a phylogenetic tree with five extant species having each a single nucleotide.

into a product having a term for each sequence position, or site,

$$f(X|\theta) = \prod_{k=1}^{N} f(X_{k\cdot}|\theta). \tag{1.3}$$

The second assumption is that the evolution in one lineage is independent from the one in other lineages.

Under these assumptions, the likelihood for a single site $f(X_{k\cdot}|\theta)$ is the sum over all possible state combinations for the extinct ancestors. Given that this likelihood is defined for a single site, the subscript k is from now on omitted as to lighten the notation. For example, the likelihood of the tree illustrated in Figure 1.2 would be defined as

$$f(X_{\cdot}|\theta) = \sum_{x_0}\sum_{x_6}\sum_{x_7}\sum_{x_8}\Big[\pi_{x_0} p_{x_0x_6}(t_6) p_{x_6x_7}(t_7) p_{x_7T}(t_1) \times \\ p_{x_7C}(t_2) p_{x_6A}(t_3) p_{x_0x_8}(t_8) p_{x_8C}(t_4) p_{x_8C}(t_5)\Big].$$

where x_i represents the state at ancestral node i, $p_{x_ix_j}(t)$ the probability of transitioning from state x_i to state x_j after t expected substitutions and π_{x_0} is the prior probability of observing states x_0 defined as the equilibrium state frequencies under the model.

Using the second assumption regarding the independence between lineages, this site-based likelihood can be rewritten using Felsenstein's *pruning algorithm* [33] stating that computation may be economized "*by trying to move summation signs in as far right as possible and enclose them in parentheses where possible*",

$$f(X_{k\cdot}|\theta) = \sum_{x_0}\pi_{x_0}\Big\{\sum_{x_6} p_{x_0x_6}(t_6)\Big[\Big(\sum_{x_7} p_{x_6x_7}(t_7) p_{x_7T}(t_1) p_{x_7C}(t_2)\Big) p_{x_6A}(t_3)\Big]\Big\} \times \\ \Big[\sum_{x_8} p_{x_0x_8}(t_8) p_{x_8C}(t_4) p_{x_8C}(t_5)\Big].$$

This representation of the likelihood has the particularity of having the parenthesis pattern matching the state occurrences in leaf nodes :

$$[(T, C), A], [C, C].$$

Indeed, this matching appears because the essence of this *pruning algorithm* is to compute successively the probability of data on many subtrees.

A more general recursive definition can be given by considering $L_{ki}(x_{ki})$ to be the probability of observing states at position k of the sequence at the descendants of node i, given that the symbol observable at node i is x_{ki}. Whenever x_{ki} is equal to the whole set of possible states $\mathcal{K}$, this conditional probability $L_{ki}(x_{ki})$ is more commonly called the conditional probability vector (CPV) for node i at position k.

The conditional probability $L_{ki}(x_{ki})$ of a node without children, a leaf node, is defined as 1 if the state x_{ki} is observed and 0 otherwise. However, if i is an internal node and is the ancestor of nodes j and l, the conditional probability $L_{ki}(x_{ki})$ is then defined as

$$L_{ki}(x_{ki}) = \left[\sum_{x_{kj}} p_{x_{ki}x_{kj}}(t_j) L_{kj}(x_{kj})\right] \times \left[\sum_{x_{kl}} p_{x_{ki}x_{kl}}(t_l) L_{kl}(x_{kl})\right]. \tag{1.4}$$

Therefore the conditional probability $L_{ki}(x_{ki})$ is built on probabilities coming from both subtrees rooted by node j, and l respectively. Each bracket represents the probability of observing a state x_{ki} transitioning to a state x_{kj} at node j, respectively x_{kl} at node l over a period of time t_j, respectively t_l. Due to the assumption of independence between lineages, the product of those probabilities defines the joint probability of observing both transitions and thus define the conditional probability $L_{ki}(x_{ki})$.

Using this scheme, the likelihood of site k is defined using the conditional probability $L_{k0}(x_{k0})$ of the root node 0,

$$f(X_{k\cdot}|\theta) = \sum_{x_{k0}} \pi_{x_0} L_{k0}(x_{k0}), \tag{1.5}$$

Based on this description of the likelihood, a rough approximation of its computational complexity can be derived for a phylogeny having S extent species defined each by molecular sequences composed of N sites having K possible states. For such rooted binary trees, the amount of internal nodes, representing ancestral species, amounts to $S - 1$ and thus the total number of nodes amounts to $2S - 1$.

For each node in the tree with the exception of the root, this likelihood requires one matrix exponentiation for $P(t) = e^{Qt}$ costing[1] roughly $\mathcal{O}(K^3)$ if eigendecompositions are employed for this computation [86]. These matrices $P(t)$ are then used at each tree nodes to compute the CPVs for each site giving a complexity[2] of $\mathcal{O}(NK^2)$. Therefore the *worst-case* likelihood complexity is defined as

$$\mathcal{O}(2S(K^3 + NK^2)), \tag{1.6}$$

[1]Known complexity for the QR algorithm employed for eigendecompositions.

[2]Known worst complexity for N matrix-vector multiplications having a complexity $\mathcal{O}(K^2)$ with a square matrix of size K.

making it far more expensive to compute than the $O(N)$ of the `PyRate` model.

The likelihood function described in this section forms the base of more complex phylogenetic analysis requiring the use of several classes of evolutionary models. Some of those models are detailed in the following sections.

1.3.4. Detecting positive selection on protein coding genes

Models of codon-substitutions have been widely used to study and identify positive selection on protein coding genes. The base model [44] defines the evolution of codons by monitoring two specific types of substitutions. The first one affects the nucleotides forming a codon and express the ratio κ of transitions ($A \leftrightarrow G$ or $C \leftrightarrow T$) to the other possible nucleotide-substitutions, the transversions.

The second type aims to differentiate non-silent codon-substitutions changing the amino acids sequence from the silent substitutions keeping it intact. Such modifications of the amino acids sequence are of key interest given that they result in biological changes in the organism. Therefore the ratio of non-synonymous (*non-silent*) to synonymous (*silent*) substitutions $\omega = dN/dS$ is a key indicator of selective pressure on codon sites.

This evolutionary model is characterised by its instantaneous substitution-rate matrix Q having rates q_{ij}, with $i \neq j$, defined as

$$q_{ij} = \begin{cases} 0, & \text{if } i \text{ and } j \text{ differ at more than one nucleotide,} \\ \pi_j, & \text{if } i \text{ and } j \text{ differ by a synonymous transversion,} \\ \kappa\pi_j, & \text{if } i \text{ and } j \text{ differ by a synonymous transition,} \\ \omega\pi_j, & \text{if } i \text{ and } j \text{ differ by a non-synonymous transversion,} \\ \omega\kappa\pi_j, & \text{if } i \text{ and } j \text{ differ by a non-synonymous transition.} \end{cases}$$

This matrix Q forms the base of a variety of evolutionary models [137].

As an example, the *M2a* model estimates the selective pressure on codon sites through the use of a mixture of three site-classes having different synonymous to non-synonymous substitution rates Ω : deleterious ($\omega_0 < 0$), neutral ($\omega_1 = 1$) and positive ($\omega_2 > 1$). Beside the estimation of the parameter values for each site-class ω_k and their respective proportions p_k, this model estimates the overall transition-transversion rate κ and the branch lengths of the tree $\mathbf{t}$. The likelihood of such a model for site k is given by

$$f(X_{k\cdot}|\kappa, \Omega, \mathbf{p}, \mathbf{t}) = \sum_{k=1}^{K} p_k f(X_{k\cdot}|\kappa, \omega_k, \mathbf{t}), \tag{1.7}$$

with $\mathbf{p}$ representing the vector of proportions (p_1, p_2, p_3).

A more advanced model is offering the possibility to estimate the selective pressure along specific sites and lineages. This branch-site models define a set of foreground branches having the same three site-classes as the *M2a* model. The remaining background branches only dispose of two site-classes representing deleterious and neutral evolutions. The possible site-classes I_k and their combinations of ω are defined in table 1.1.

Table 1.1.: Site-classes I_k and their combinations of ω for the branch-site model.

Site-class	Proportion	Background ω	Foreground ω
0	p_0	$0 < \omega_0 < 1$	$0 < \omega_0 < 1$
1	p_1	$\omega_1 = 1$	$\omega_1 = 1$
2a	$(1 - p_0 - p_1)p_0/(p_0 + p_1)$	$0 < \omega_0 < 1$	$\omega_2 > 1$
2b	$(1 - p_0 - p_1)p_1/(p_0 + p_1)$	$\omega_1 = 1$	$\omega_2 > 1$

The likelihood function of this model is averaged over the different classes similarly as in the *M2a* model,

$$f(X_{k\cdot}|\kappa, \Omega, \mathbf{p}, \mathbf{t}) = \sum_{k=1}^{K} p_{I_k} f(X_{k\cdot}|\kappa, \omega_{I_k}, \mathbf{t}), \tag{1.8}$$

where p_{I_k} define the proportions for class I_k and ω_{I_k} encompasses the foreground and background ω.

Other approaches offering similar models have been proposed in the literature. Random effects branch-site models [97] mainly differ from the previously described branch-site model in the mixing of the different site-classes. Instead of being integrated at the *tree-level* (Eq. (1.8)), site-classes are used to form an average probability transition matrix $\bar{P}$ at the *node-level* such that

$$\bar{P} = \sum_{C=1}^{C} p_c e^{Q_c t},$$

with p_c representing the proportions of sites in class c and Q_c the instantaneous transition-rate matrix defined using ω_c.This average transition matrix $\bar{P}$ is then used as previously to compute the CPVs.

Another model allows site-specific variations of the selection patterns along lineages [49]. This model does not specifically constrain lineages to evolve under a specific class but considers an enlarged substitution process. The extended instantaneous transition-rate matrix $S = D + R$ encompass two distinct types of continuous-time Markov processes. The first one is the previously described site-classes approach. Indeed, D is formed as a block diagonal matrix having matrices Q_c constructed using w_c on the diagonal. The second process, defined by the matrix R, enables the transition from one site-class to another one through time. Three new parameters δ, α, β are introduced in R. The first one, δ, represents the overall lineage switching rate, while α and β define the site-class pairwise direction of switches (e.g. from site-class 1 so site-class 2).

The likelihood of the aforementioned codon-substitution models is expensive for two reasons. The first one is that there are 61 possible codon-states. When compared to the only 4 states of nucleotide-substitution models, this increase in number of states K significantly augments the computational cost of CPVs and matrix exponentiations. The second reason, and probably the most important, stems from the multiple computations

of the tree likelihood (Eq. (1.5)) required for each site-classes. Indeed, this increase in computations appears clearly in equation (1.7) and equation (1.8), where the tree likelihoods associated with all possible site-classes are averaged.

1.3.5. Assessing coevolution between pairs of sequence positions

The family of models previously presented was based on the assumption that the positions in a sequence are evolving independently of each others. However, this is known as being inaccurate and the study of these jointly evolving positions is part of the concept of coevolution defined as "*the modification of a biological object triggered by the change of a related object*" [134]. Coevolving positions in molecular sequences have been shown to have a direct effect on the function and structure of proteins in multiple studies [12, 26]. Until recently this evolutionary concept has only been analysed using measures of correlation between sequences. However, recently, a molecular evolutionary model based on phylogenetic tree has been conceived [27].

This novel evolutionary model relaxes the assumption that sequence positions are evolving independently and thus revokes the use of equation (1.3). For that reason, only two sites of a sequence are considered at a time and the model represents the potential substitutions of this pair of positions considering a given profile of coevolution ϕ. This profile defines the set of possible pairs of states that the positions can take at any time. Assuming a set of aligned nucleotide sequences, such profile ϕ could define the states $\{AT, TG\}$. Coevolving positions would imply that upon a substitution from A to T in position 1, a subsequent substitution from T to G should be observed in position 2. The total amount of possible profiles [28], M, for a pair of positions having K possible states is defined as

$$M = \sum_{k=2}^{K} \left(\frac{K!}{(K-k)!} \times \frac{1}{k!} \right)^2 \tag{1.9}$$

The continuous-time Markov chain used to define this process represents pairs of states, instead of single states as in the previous models. Therefore the instantaneous substitution-rate matrix Q is composed of K^2 states and defined the $K^2 \times K^2$ possible rates q_{ij}, with $i \neq j$, as

$$q_{ij} = \begin{cases} 0, & \text{if } i \text{ and } j \text{ differ at more than one nucleotide,} \\ r_1, & \text{if } \{i,j\} \notin \phi \text{ if } i \text{ differs from } j \text{ at position 1,} \\ r_2, & \text{if } \{i,j\} \notin \phi \text{ if } i \text{ differs from } j \text{ at position 2,} \\ s, & \text{if } i \in \phi \text{ and } j \notin \phi, \\ d, & \text{if } i \notin \phi \text{ and } j \in \phi. \end{cases}$$

The parameter s is the rate of transition from a coevolving combination present in the profile to a non-coevolving combination. Conversely, d is the rate of transition from one non-coevolving to a coevolving combination. The substitution rates of non-coevolving positions are represented by parameters r_1 and r_2 and correspond to independent positions evolving according to a Jukes-Cantor model (Eq.(1.2)).

The likelihood of this model is therefore defined as

$$f(X_{\{ij\}\cdot}|s, d, r1, r2, \phi, \mathbf{t}),$$

with $\{ij\}$ defining the pair of positions i and j in the sequence and using the tree likelihood defined in equation (1.5).

The computational complexity of this model is independent of the sequence size N but increases due to the expanded states number of K^2 leading to a $\mathcal{O}(K^6)$ computational cost for the matrix exponentiation. However, when considering nucleotide sequences, with $K_{nucl} = 4$, the cost of a likelihood evaluation remains cheap when compared to the one of models based on the tree likelihood detailed in the previous sections.

Therefore the challenge arising from statistical analysis on this model is different from the one yet encountered. The expensive computational cost is caused by the large number of independent analyses required to analyse a molecular dataset formed of a phylogenetic tree and a molecular sequence per specie. Indeed, there exists $N^2/2$ distinct pairs of positions in a sequence. For each of these distinct pairs an analysis must be run, in the worst case, for M potential coevolving profiles ϕ. Therefore the total number of independent statistical analyses for a dataset amounts to $M \cdot N^2/2$. Such an amount is further doubled when comparative analyses are conducted using ML methods due to the computation of the null hypothesis. This hypothesis assumes an independent evolution of the two sites according to a Jukes-Cantor model.

1.3.6. Inferring phylogenetic trees

A fundamental challenge in phylogenetic analysis is to estimate the best phylogenetic tree supported by molecular sequences. This challenge comes from the difficulty of exploring the space of possible tree topologies. Indeed, the size of this space grows according to a factorial function of the number of leaf nodes, or extent species. As an example, there exists close to 35 millions different topologies for a rooted phylogenetic tree having 10 species [33]. Therefore exhaustive approaches cannot be used for phylogeny composed of hundreds or thousands of species.

In order to avoid the exhaustive scheme visiting all the possible trees, cheaper strategies must be employed to explore the tree space. The strategies usually used start from an initial tree and rearrange some of its branches to reach neighbouring trees. The tree space is then explored by moving from a tree to one of its neighbours and so on. Using such moves, heuristic methods can be conceived to search for the best tree.

The design of tree moves has an important impact on heuristic methods. Indeed, it is well known in the field of metaheuristics, that such algorithms require an intensification and a diversification components. The first one aims at intensifying the exploration in the vicinity of the current solution, or tree topology in our case, to refine it. Oppositely, the second tends to produce more drastic changes to diversify the exploration of the solution space and to avoid getting stuck in local optima.

Tree moves[3] based on *branch-change* aims at intensifying the search and consists in moving a node of the tree continuously along its edges as illustrated in figure 1.3a.

[3]This classification of tree moves is based on the review of Lakner et al. [76].

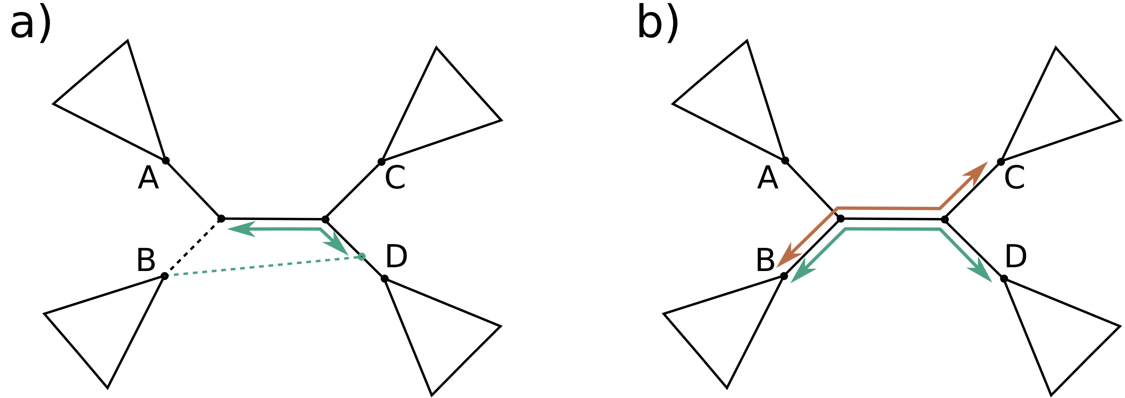


Figure 1.3.: In these figures illustrating different tree moves, triangles are employed to represent subtrees. The left figure illustrates a branch-change, while the right one a Stochastic Nearest Neighbour Integerchange.

While such moves usually perform small slides, they can sometimes produce changes in the topology whenever a node slide across another one.

Tree moves using *branch-rearrangement* are further divided into two categories : *swapping* and *pruning-regrafting* moves. As defined by their denomination, the first category swaps two subtrees. One of these swapping move, the *Stochastic Nearest Neighbour Interchange* (STNNI) is represented in figure 1.3b. Swapping moves cause greater changes in the tree topology than *branch-change* moves and are therefore more diversifying.

The prunning-regrafting move first prunes a subtree and then regrafts it on another randomly chosen branch of the tree. One of these moves, Subtree Pruning and Regrafting (SPR), is shown in figure 1.4 where the subtree rooted by node A is regrafted between node D and E. Changes produced by such procedure is further more diversifying than the two previous ones.

When inferred in a statistical framework, the analysis of the phylogenetic tree is guided by the tree likelihood (Eq. 1.5) and requires, jointly with the already challenging search of the best tree topology, the estimation of the evolutionary model parameters θ. Furthermore, any change in the tree topology modifies the likelihood function and requires θ to be re-estimated.

Under these circumstances, these analyses require a significant amount of likelihood evaluations. Therefore, simple evolutionary models are preferred in order to reduce the overall computational cost. One of the most widely used model for such analyses is the general time reversible (GTR) substitution model. This nucleotide model uses six parameters to define the instantaneous substitution rate matrix Q, as

$$Q_{GTR} = \{q_{ij}\} = \begin{bmatrix} \cdot & a\pi_C & b\pi_A & c\pi_G \\ a\pi_T & \cdot & d\pi_A & e\pi_G \\ b\pi_T & d\pi_C & \cdot & f\pi_G \\ c\pi_T & e\pi_C & f\pi_A & \cdot \end{bmatrix}$$

with (a, b, c, d, e, f) defining the transition rates from one nucleotide to the other and π_j

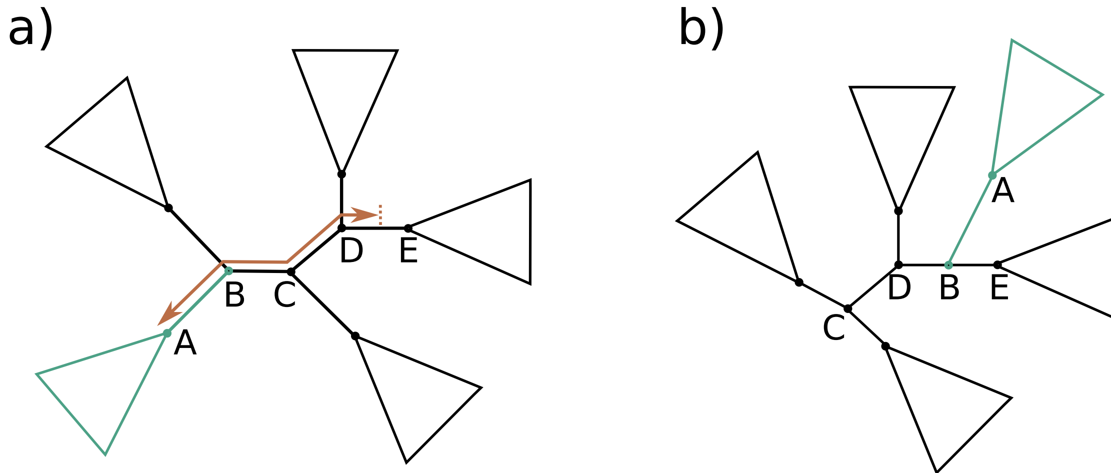


Figure 1.4.: Illustration of a Subtree Pruning and Regrafting move. The left figure shows the subtree pruned and its grafting point. The state of the tree after this move is represented in the right figure.

the equilibrium frequency of nucleotide j.

Using the aforementioned tree moves and these simplistic nucleotide models, the best phylogenetic tree can be estimated. However, this optimal tree topology does not necessarily provide any confidence level to use the obtained topology for further analysis. While such confidence in the tree topology can be readily derived from posterior distributions obtained during Bayesian analysis, additional steps must be enforced for ML methods. Indeed, confidence, or support, on the tree features, can be built using statistical *bootstrapping* at the cost of supplementary likelihood evaluations.

Consequently, the computational complexity of inferring phylogeny quickly grows with the number of species S given that it impacts both the computational costs of the tree likelihood and the size of the tree topology space. Indeed, as the tree space grows, the amount of likelihood evaluations required for its exploration increases. For this reason, this application provides a different challenge than the detection of positive selection on protein coding genes.

2. Model representation and likelihood computation

The representation of statistical models and the computation of their likelihood are the cornerstone of HOGAN. Indeed, this representation must, first and foremost, enable the definition of the existing phylogenetic-based models but also the upcoming ones and, within the realm of the possible, any models. While this already poses a serious challenge, this key element must offer generic functionalities that improve the computational efficiency of likelihood evaluations.

This chapter begins by exposing why using *Directed Acyclic Graphs* (DAG) fulfils all these requirements. This introduction is followed by a brief definition of this specific type of graph as well as an illustration on the phylogenetic tree likelihood. The two key features, *partial likelihood updates* and *parallel likelihood evaluations*, that generically improve the computational efficiency of likelihood evaluations are then described. The behaviour of both methods is analysed on the branch-site model defined in equation (1.8) and their performance is compared with a state of the art implementation of the model in question.

2.1. Why directed acyclic graphs ?

Graphs are a widely used mathematical formalization of networks as in the sense of sets of objects having potential interactions with each others. This formalization offers numerous advantages starting by its visual representation. Indeed, graph visualization offers an intuitive and universal way of representing relations between objects and is the subject of a field of research, *graph drawing*, combining geometric graph theory and information visualization [25].

More importantly, graph theory, the study of graphs, has been and remains an active research field. Numerous graph properties and algorithms developed in this field are available for problems such as routing problems or graph partitioning problems. These problems are tightly coupled with theoretical computer science problems such as the Travelling Salesman Problem (TSP [7]) or the scheduling of parallel computations [22]. This last application highly motivated the choice of directed acyclic graphs for the representation of statistical models.

Indeed, parallel computation in programs are commonly expressed as the graph of dependencies, or precedences, between their inner computational tasks. Under the condition that such program does not contain deadlocks, its dependency graph forms a directed acyclic graph. The scheduling of such a directed task graph on multiprocessors

has been widely studied and, despite being a NP-Complete problem, multiple efficient heuristics exist for it [74].

A second motivation for the use of DAG comes from the concept of *graphical models* [64] developed in computational statistics. Indeed, the representation of statistical applications in this field has become of growing importance considering the emergence of models having thousands or billions of random variables interacting in complex ways. In order to tackle this challenge, directed or undirected graphs are used to represent families of probability distributions.

In addition of offering a unifying framework for the description of statistical models, this use of graphs has built a bridge between statistics and computer science. Indeed, based on this novel definition, numerous inference algorithms have been developed to reduce the computational complexity of inferring such models [131, 68]. One of the most widely used MCMC sampler in computational statistics, BUGS [80], is directly based on the concept of graphical models and its development highly contributed to the field.

While the initial idea of applying graphical models for phylogenetic-based models emerged in 2004 [64], it had to wait until the end of 2014 to reach the evolutionary biology community and be proposed as a unifying and standardised representation of such models [59]. Using this formalization, the authors of this latter article implemented and distributed a software for Bayesian phylogenetic inference, `RevBayes`. This software offers an easy access through the use of `Rev`, a custom R-like interpreted language, to algorithms for graphical models such as the *Sum-Product* algorithm [38, 64], more known, in phylogenetics, as the Felsenstein's *pruning algorithm* [33] and MCMC methods.

In conclusion, the choice of DAGs as the cornerstone of `HOGAN` is motivated by their potential to represent parallel computations in programs as well as graphical models. Both of these concepts offer an intuitive representation of a likelihood from the statistical and computational point of view and also address the framework need for efficient and generic algorithms dedicated to likelihood evaluations. Finally, as demonstrated in the following chapters, the graph representation of a likelihood evaluation serves as a basis for the formalization and the development of model-independent optimization strategies.

2.2. A short introduction to DAGs

A directed graph $\mathcal{G}$ defines an ordered pair $(\mathcal{V}, \mathcal{A})$ where

- $\mathcal{V}$ is the finite non-empty set of vertices v, or nodes, of the graph;
- $\mathcal{E}$ is the finite set of ordered pairs e of vertices defining the directed edges of the graph.

The directed edge between nodes v_s and v_t is defined as the pair $e = (v_s, v_t)$ having v_s as tail and v_t as head.

A *directed walk* in $\mathcal{G}$ is a finite non-null sequence of $W = v_0e_1v_1e_2...e_kv_k$, or $W = v_0v_1...v_k$, with directed edge e_i having node $i-1$ as tail and node i as head. A *directed walk* is *closed* if the origin v_0 and terminus v_k nodes are the same.

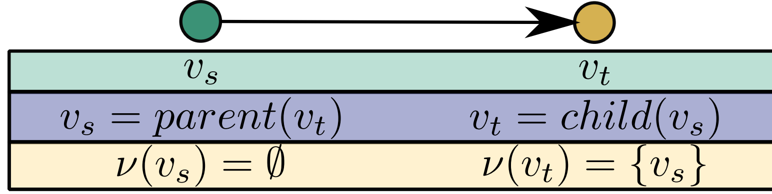


Figure 2.1.: Example of the relation between nodes, or vertices, v_s and v_t.

A *directed trail* is a directed walk W where all directed edge e_i are distinct. A closed directed trail whose origin and internal vertices are distinct is a directed cycle. Therefore, a *directed acyclic graph* is a directed graph having no directed cycle.

A node v_s is defined as *parent* node of v_t if the edge $e = (v_s, v_t)$ exists in $\mathcal{E}$. For each node $v \in \mathcal{V}$, the subset of indices $\psi(v)$ defines the indices of the parent nodes of v and the subset of nodes $\nu(v)$ defines the parent nodes of v.

2.2.1. Directed task graph

Figure 2.2a) illustrates an example of a DAG explaining a program by its task dependencies. The program is composed of tasks $\{v_1, v_2, v_3, v_4, v_5\}$ represented as red and yellow nodes and input variables $\{v_A, v_B\}$ represented as green nodes. The parent nodes of node v_A are $\nu(v_A) = \{v_2, v_3, v_4\}$ with indices $\psi(v_A) = \{2, 3, 4\}$. These parent nodes $\nu(v_A)$ depends on the results of v_A for their computations.

Correct executions represented by directed task graphs can be identified by the topological sorting of their vertices. Such ordering defines that node v_t precedes node v_s if there exists an edge $e = (v_s, v_t) \in \mathcal{E}$. According to this definition, correct executions, or *DAG traversal*, of the program represented in figure 2.2a) could be expressed by the sequences $(v_A, v_B, v_2, v_3, v_4, v_5, v_1)$ or $(v_A, v_2, v_3, v_B, v_4, v_5, v_1)$.

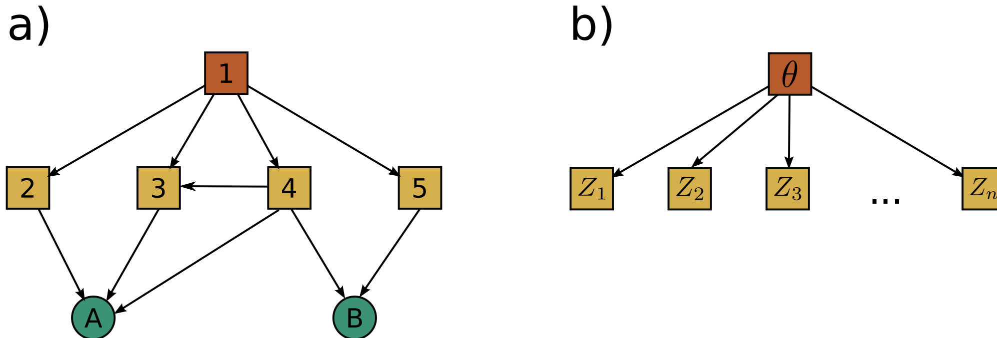


Figure 2.2.: Two examples of DAGs. Left figure show a simple DAG. Right figure show a DAG representing a graphical model.

Assuming that more information was known such a DAG could be weighted. For example, edges $e = (v_s, v_t) \in \mathcal{E}$ could be weighted in function of the communication cost between the tasks v_s and v_t. For example, edge $e = (v_4, v_A)$ could be weighted to

8 (bytes) assuming that task v_4 would require a single integer from task v_A. Similarly, node v_s could be weighted by the computational cost, or time, of its designated task χ_s. For example, if the task identified by subscript 4 and represented by node v_4 would take 7 seconds to be computed, then $\chi_4 = 7$.

2.2.2. Graphical model

Figure 2.2b) illustrates a DAG representing a graphical model [131]. Each node $v \in \mathcal{V}$ of such DAG are identified with random variables X_v taking values $x_v \in \mathcal{X}_v$. Assuming a collection of discrete or continuous kernels $\left\{k\left(x_v|x_{\nu(v)}\right) : v \in \mathcal{V}\right\}$ summing, or respectively integrating to 1, the joint probability distribution represented in this type of DAGs is given by

$$p(x_{\mathcal{V}}) = \prod_{v \in \mathcal{V}} k\left(x_v|x_{\nu(v)}\right).$$

From this expression, it clearly appears that the kernels $k\left(x_v|x_{\nu(v)}\right)$ are the conditionals probability distributions of $p(x_{\mathcal{V}})$ and can therefore be defined as

$$k\left(x_v|x_{\nu(v)}\right) = p\left(x_v|x_{\nu(v)}\right).$$

Using these definitions, the DAG of figure 2.2b) describes N conditionally independents variables Z_i with values z_i identically distributed in function of parameter θ. Therefore, the joint probability of such a graphical model is then defined as

$$p(x_{\mathcal{V}}) = p(\theta) \times \prod_{i \in N} p\left(z_i|x_{\nu(z_i)}\right) = p(\theta) \times \prod_{i \in N} p\left(z_i|\theta\right).$$

2.2.3. Phylogenetic tree likelihood

The phylogenetic tree likelihood defined by equation (1.5) is the core of phylogenetic-based models. Statistical analyses conducted on such models require thousands to billions of likelihood evaluations. Therefore, designing an efficient implementation of the phylogenetic tree likelihood is of utmost importance.

Implementation as a directed task graph

In `HOGAN`, this likelihood is designed as a directed task graph. An example of such graph for a phylogeny having six species is shown in figure 2.3. The notion of likelihood defined in equations (1.4) and (1.5) has to be slightly rewritten to better explain this graph and to reflect more accurately the computations linked to each of its tasks.

Indeed, the likelihood function can be redefined using linear algebra expressions. Assuming that all possible states $\mathcal{K}$ are always considered for x_{ki}, even if their probability is null, and this for each sequence position k and node i, then the conditional probability vectors g_{ki} can be used to redefine Equation (1.4) as

$$g_{ki} = L_{ki}(x_{ki}) = \left[P(t_j)g_{kj}\right] \circ \left[P(t_l)g_{kl}\right] = h_{kj} \circ h_{kl} \tag{2.1}$$

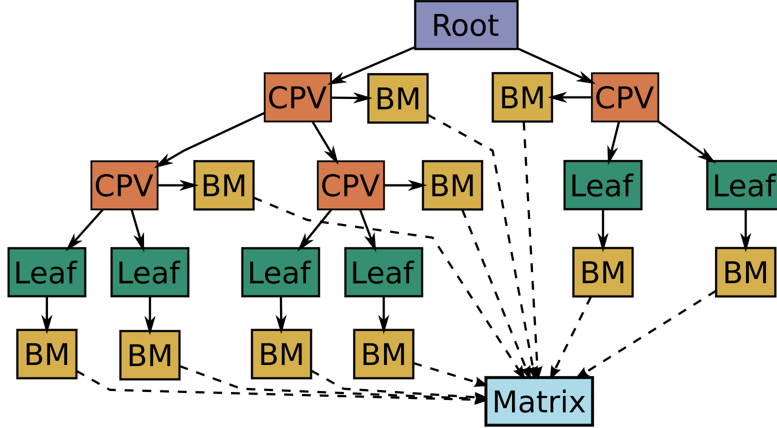


Figure 2.3.: Directed task graph representing a generic phylogeny-based likelihood. The six green nodes, entitled leaf, contain the molecular data of the extent species. Using the *Matrix* and *BM* nodes, the CPVs are computed from the leaf nodes upto the root node.

with $\circ$ defining the Hadamard product of two vectors or matrices.

Most of the phylogenetic-based software, such as `FastCodeML` or `MrBayes`, are using the expression defined in Eq. (2.1) to compute the CPVs. However, this approach makes an extensive use of matrix-vector multiplications, while matrix-matrix multiplications could, and should, be used. Indeed, matrix-matrix multiplications offer better computational performance, due to memory optimizations, than multiple calls to their matrix-vector counterparts.

Matrix-matrix multiplications are applied by compacting conditional probability vectors in a single matrix at each node. Given the compacted CPVs matrix,

$$G_i = [g_{1i} ... g_{ki} ... g_{Ni}],$$

for node i containing the CPVs of the N sequence positions, Eq. (2.1) can be redefined as

$$G_i = \Big[P(t_j)G_j\Big] \circ \Big[P(t_l)G_l\Big] = H_j \circ H_l. \tag{2.2}$$

Equations (1.3) and (1.5) are then adapted to define the final likelihood as

$$f(X|\theta) = \prod_{k=1}^{N} \left(\pi_{x_0}^T G_0\right)_k \tag{2.3}$$

with $(\cdot)_k$ representing the kth element of the vector representing the site-wise conditional probabilities previously defined by equation (1.3).

Another fundamental step in the computation of the likelihood is the matrix exponentiation required to compute the transition probability matrices $P(t)$ (Eq. 1.1). This crucial step is done using an eigendecomposition of the instantaneous substitution-rate matrix Q [86, 127] as

$$Q = L\Lambda L^{-1}. \tag{2.4}$$

with L as the left eigenvectors and Λ as the diagonal matrix of eigenvalues. Using this eigendecomposition, the $P(t)$ matrix is then obtained according to

$$P(t) = e^{Qt} = Le^{\Lambda t}L^{-1}. \tag{2.5}$$

Using the previous definitions, the directed task graph of the likelihood illustrated in figure 2.3 can now be properly detailed. The different tasks, ordered by their appearance in the topological ordering of nodes, are defining the following computations :

- The **Matrix** task from the instantaneous substitution-rate matrix Q based on parameters θ. A first step toward the computation of the transition probability matrices $P(t)$ is also conducted by applying the eigendecomposition of Q (Eq. (2.4)).
- The branch-matrix (**BM**) tasks compute each probability transition matrices $P(t)$ according to equation (2.5), in function of the branch length t defined by the parameters θ.
- The **Leaf** and **CPV** tasks employ the $P(t_i)$ matrix computed in branch-matrix tasks to define their respective H_i matrix (Eq. (2.2)). Leaf tasks compute H_i in function of the molecular data $X_{\cdot i}$ of their respective extent species as

  $$H_i = P(t_i)G_i = P(t_i)X_{\cdot i}.$$

  CPV nodes compute H_i in function of their child nodes j and l as

  $$H_i = P(t_i)G_i = P(t_i)(H_j \circ H_l).$$

- The **Root** task computes the final likelihood value according to equation (2.3).

Advantages of the proposed implementation

This task decomposition offers several advantages. The first is that the costly part of the matrix exponentiation, the eigendecomposition of the matrix Q, is only applied once. Then, the computation of the probability transition matrix $P(t)$ is only applied once per pair of branch and Q matrix. For site-classes or branch-site models, this representation ensures that these costly computations are only computed once. For example, the branch-site model defined in table 1.1 would only require the computation of one matrix node defined in function of ω_0 for background site-classes $\{0, 2a\}$ and foreground site-class 0.

Another aspect is the notion of task granularity that plays an important role in parallel programs. The decomposition presented here has a coarse granularity for leaf, CPV and root nodes, in the sense that the computation of all CPVs are compacted in a single node by using equation (2.2). However the granularity of these nodes can be easily tuned by dividing the set of sequence positions in several subsets that would each be computed by dedicated tasks, or nodes.

For example, to get a granularity twice finer, two subsets of positions defined as $N_1 = [1..N/2)$ and $N_2 = [N/2..N]$ could be assigned to two different sub-DAGs, each of them

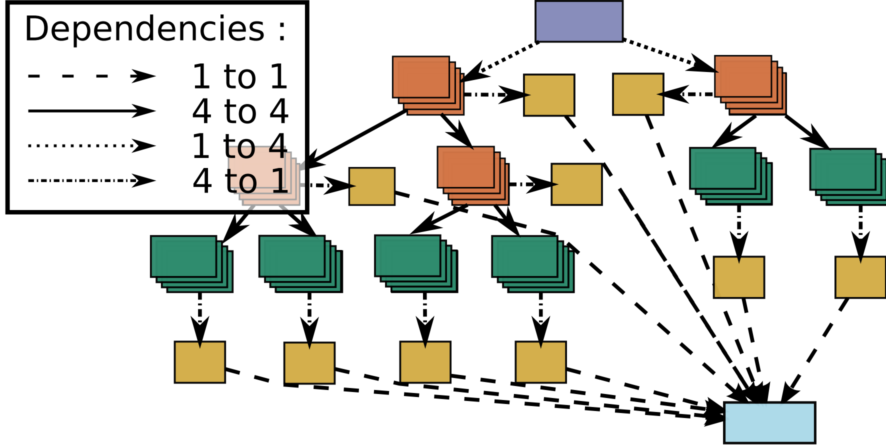


Figure 2.4.: Directed task graph representing a generic phylogeny-based likelihood with a fined granularity. The sequence positions have been separated in four different subsets. Edges may represent multiple dependencies.

based on equation (2.2), with their respective sub-matrices $G_{N_1 i} = \{g_{ki} : k \in N_1\}$ and $G_{N_2 i} = \{g_{ki} : k \in N_2\}$. The final likelihood would be obtained by using equation (2.3) with $G_0 = [G_{N_1 0} G_{N_2 0}]$. In order to refine the granularity, this scheme could be applied as much as required until reaching subsets containing a single CPV per node; situation corresponding to equation (2.1).

Refining the granularity of the leaf and CPV nodes does not require additional matrix and branch-matrix nodes. Therefore, no additional computations would be required since the same nodes would be reused. However, new edges would be added to the branch-matrix nodes in order to enforce the dependencies of the newly created leaf and CPV nodes. An example of such scenario is showed in figure 2.4 with four different sub-DAGs.

Advantages over graphical models

The directed task graph representation differs from the unifying representation based on graphical models proposed by Höhna *et al.* [59]. Graphical models identify the conditional dependencies between stochastic variables that, in the present case, mostly represents the CPVs. By explaining the underlying computations of these stochastic variables as well as their dependencies, directed task graphs offer more power to optimize the computational efficiency of a likelihood given that the granularity of the DAG is not constrained to identify the stochastic processes.

Given that the directed task graph is a refined version of a graphical model, interfacing the first with the second is straightforward. A simplified illustration of such an approach is shown in figure 2.5 where the computational tasks of figure 2.3 are aggregated into nodes representing the stochastic variables of a graphical model. The graphical model obtained matches thus the unifying representation of Höhna *et al.* [59].

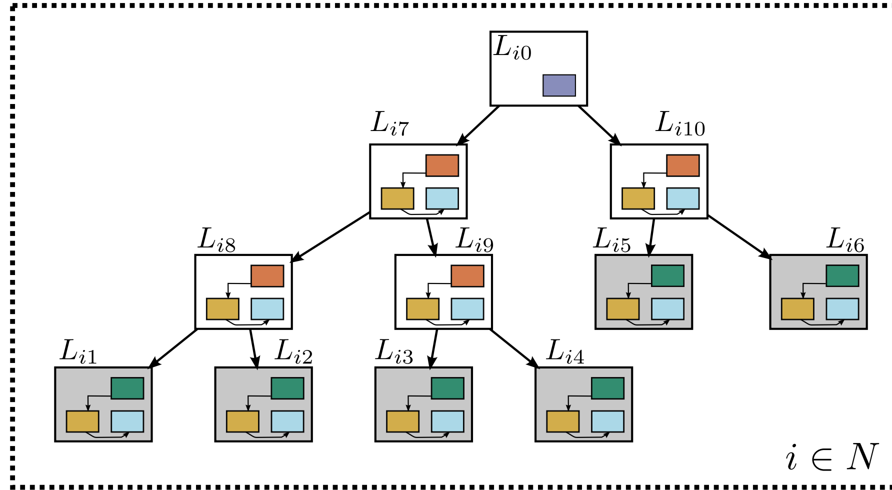


Figure 2.5.: Graphical model representing a generic phylogeny-based likelihood. The inner graph in nodes links the graphical model with the directed task graph of figure 2.3. Nodes having white background identify stochastic nodes, while the ones with grey background identify clamped nodes. The dotted rectangle identifies a *plate*, a structure representing iterations.

Therefore, directed task graphs are better fitted to `HOGAN` which is conceived for computational efficiency. In addition of having the power to be easily converted to the graphical model representation, they enable more freedom to optimise the likelihood evaluations.

2.3. Partial Likelihood Update

Methods employed for statistical analysis, such as ML or MCMC methods, have to explore the parameter space for maximization or for integration purpose. Strategies employed for this matter move through the parameter space by updating the current parameters $\theta = (\theta_1, .., \theta_i, .., \theta_d)$ to a new set of values $\phi = (\phi_1, .., \phi_i, .., \phi_d)$. A move updating all parameters such that

$$\theta_i \neq \phi_i \quad \forall \theta_i \in \theta \tag{2.6}$$

defines a *full likelihood evaluation* (FLE). By opposition, moves only updating a subset of parameters identified by the set $\mathcal{A}$ such that

$$\theta_i \neq \phi_i \quad \forall \theta_i \in \mathcal{A} \text{ with } \mathcal{A} \subseteq \theta \tag{2.7}$$

defines *partial likelihood updates* (PLU). This last type of move is frequently used in both aforementioned statistical approaches for separate reasons that are detailed in chapter 3 and 4.

In the event that parameters identified by subscript $i \in \mathcal{A}$ solely impact a specific part of the likelihood, then PLUs could be computed at a cheaper cost that FLEs. Indeed, in such a case, only the impacted parts of the likelihood should be recomputed, while the remaining parts would be already available from the previous likelihood evaluation. Reusing these previously computed partial results implies that they are stored and therefore comes at the cost of memory space. Most model-specific software, such as `MrBayes` and `FastCodeML`, make use, to a certain extent, of this concept.

Using DAGs to represent likelihood computations enable the generalization of this approach for any model. Indeed, `RevBayes`, through the use of graphical models [59], proposes a generalization in regard of the conditional probabilities expressed by nodes. However, this splitting of the likelihood function does not offer much freedom in regard of which partial results can be reused during the likelihood updates. The finer likelihood representation based on directed task graphs employed in `HOGAN` offers a greater potential for that matter, given that the nodes truly represent computational tasks.

Algorithm 6 Partial likelihood update

`1) Define nodes directly impacted`
$V = \bigcup_{\theta_i \in \mathcal{A}} [\zeta(\theta_i)]$
$\Psi_{\mathcal{A}} = V$
while $V \neq \emptyset$ **do**
  `2) Define nodes indirectly impacted`
  $V = \bigcup_{v \in V} [\nu(v)]$
  $\Psi_{\mathcal{A}} = \Psi_{\mathcal{A}} \cup V$
end while
return $\Psi_{\mathcal{A}}$

Based on the DAG formalism, the set of nodes requiring to be recomputed for PLU can be easily defined. Indeed, the first nodes requiring to be reprocessed are the ones directly impacted by the change of parameters $\theta_i \in \mathcal{A}$ and are defined as the set of nodes $\zeta(\theta_i)$. This first set is then used to define the set encompassing all the nodes to recompute, $\Psi_{\mathcal{A}}$, by considering the dependencies expressed in the directed task graph as defined in algorithm 6. This operation can be combined with the mandatory DAG traversal applied during FLEs and thus has nearly no additional computational cost.

For instance, PLUs are particularly adapted to phylogenetic-based models. Considering only PLUs occurring when a single parameters θ_i is changed, such that $|\mathcal{A}| = 1$, then two different scenarios can be identified:

1. an update of an evolutionary model parameter, requiring the matrix Q to be recomputed;

2. an update of a branch length parameter, requiring CPVs to be recomputed.

In the first case, the PLUs offer no improvements given that upon the update of the Q matrix the whole likelihood must be re-evaluated which is equivalent to a FLE. Indeed, all the nodes in the DAG depend, directly or indirectly, on the Matrix node as illustrated

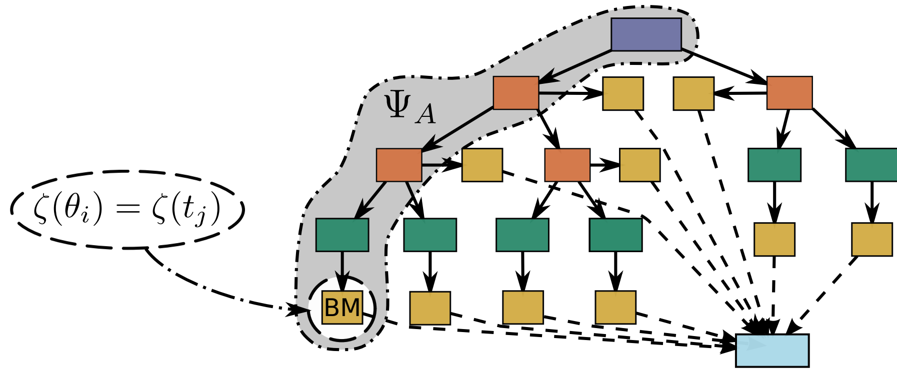


Figure 2.6.: Illustration of a partial likelihood update. The change of a branch length t_j directly impacts its dedicated BM node, $\zeta(t_j)$, and indirectly the nodes depending on it. This set of nodes that must be recomputed $\Psi_{\mathcal{A}}$ is highlighted by a grey halo encompassing all of them.

in figure 2.3. However, whenever only a branch length is updated, then partial updates become particularly efficient as illustrated in figure 2.6.

Such PLU, that modifies one branch length t_j, requires only the evaluation of the related matrix $P(t_j)$, or BM node, as well as all dependent Leaf/CPV/Root nodes. Therefore, given that a rooted phylogenetic tree is a full binary tree, the amount of these dependent nodes has a higher bound defined by the height h of the tree. In the best-case, the tree is balanced, or perfect as in figure 2.7a) implying that $h_b = log_2(S) - 1$ with S representing the number of leaves, while in the worst-case the tree is totally unbalanced as in figure 2.7b) implying that $h_w = S - 1$.

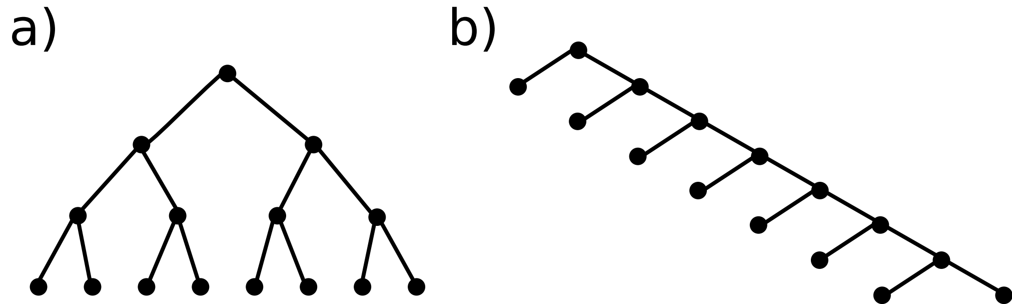


Figure 2.7.: Left figure shows a balanced full binary tree, or perfect binary tree. Right figure shows a unbalanced full binary tree that has the maximal height in function of the number of leaf nodes.

Given that there is one branch length per edge of the tree, the average amount of computations for the previously described PLUs is bounded by the average height of

worst-case tree $\bar{h}_w$ and the one of the best-case tree $\bar{h}_b$, that are defined as

$$\bar{h}_w = \frac{\sum_{i=1}^{h_w} 2i}{2S-1} = \frac{\sum_{i=1}^{h_w} 2i}{2h_w+1} = \frac{(h_w+1)^2-(h_w+1)}{2h_w+1}, \tag{2.8}$$

$$\bar{h}_b = \frac{\sum_{i=1}^{h_b} i2^i}{2S-1} = \frac{\sum_{i=1}^{h_b} i2^i}{2^{h_b+1}-1} = \frac{2^{h_b+1}(h_b-1)+2}{2^{h_b+1}-1}. \tag{2.9}$$

Assuming large heights h_w and h_b, then the average heights would be proportional to

$$h_w \gg 1 \rightarrow \bar{h}_w \approx h_w/2 = S/2, \tag{2.10}$$

$$h_b \gg 1 \rightarrow \bar{h}_b \approx h_b = \log_2(S) - 1. \tag{2.11}$$

Therefore, the average complexity of such PLUs, $\mathcal{O}_{\text{PLU}}$, is then reduced from the complexity of a FLE given by equation (1.6))

$$\mathcal{O}(2S(K^3 + NK^2)),$$

to

$$\mathcal{O}(K^3 + \log(S)NK^2)) \leq \mathcal{O}_{\text{PLU}} \leq \mathcal{O}(K^3 + SNK^2),$$

using the previously defined average tree heights as lower and upper bounds.

2.3.1. Improvements over `FastCodeML`

To illustrate the potential of partial updates on a likelihood represented as directed task graphs, `HOGAN` is compared to a state of the art implementation of the branch-site model defined in equation (1.8), `FastCodeML` [127]. This model-specific implementation improves the original implementation of this model, `CodeML` [135], by using more efficient matrix exponentiations and by using a form of compression compacting subtrees having identical CPVs over different sequence positions; later referred as *subtree compression*.

While these improvements are also implemented in `HOGAN`, both implementations differ in some aspects. First, `HOGAN` uses compacted CPVs matrices to take advantage of cache effects and memory reuse (Eq. (2.2)), while `FastCodeML` applies the computation separately on CPVs as in equation (2.1). Second, `FastCodeML` uses a coarse representation of the computational tasks that can be reused during PLUs. Indeed, as illustrated in figure 2.8, only two tasks are defined: the computation of the Q matrix and the computation over the whole phylogenetic tree. As described previously, `HOGAN` disposes of a more finer representation of tasks.

As to analyse the potential performance gains coming from these improvements, both software were compared over three different settings aiming to identify the contribution of each of them. On the first two settings, the subtree compression was disabled to ensure that the measures accurately reflected the effect of the sequence size, N, on the performance. The three settings are defined as follow :

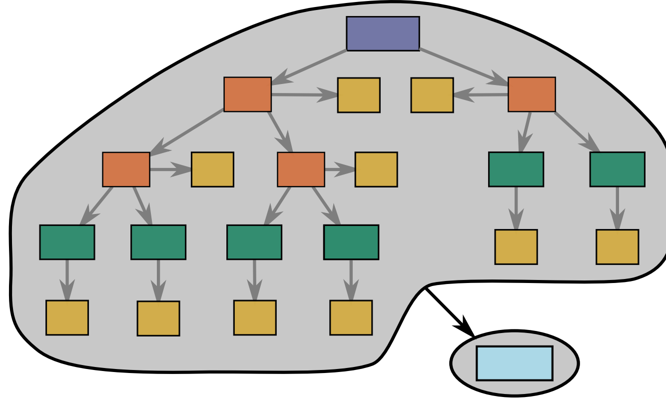


Figure 2.8.: DAG representation of the PLUs in `FastCodeML`. Only two different tasks are *hard-coded*. One for the matrix computation and one for the remaining computations

- The first setting consisted in comparing both implementations under a similar configuration in regard of the task granularity. For that matter, `HOGAN` was forced to use a coarser granularity identical to the one of `FastCodeML`, illustrated in figure 2.8.
- In the second setting, this constraint on the task granularity was relaxed to highlight the full potential of directed task graphs at reusing partial computations during PLUs.
- The final setting was based on the previous one with the exception that the subtree compression was enabled. This last step was applied in order to insure that the performance improvements previously observed were still present with subtree compressed.

For each setting, both software were compared on a dataset simulated with the software `INDELible` [36]. This dataset was composed of three different replicates of 36 simulated phylogenetic trees and sequence alignments to robustly measure the performance on problems having various sizes. For that matter, five phylogenetic trees were simulated with respectively 16, 32, 64, 128, 256 and 512 taxa and, for each of these trees, three different replicates of codon sequences were generated with respectively 10, 25, 50, 100, 250 and 500 positions.

Finally, for each software and instance in the dataset, the set of all possible PLUs induced by single parameter change, defined according to equation (2.7) as

$$\{\mathcal{A}_i = \{\theta_i\} : \theta_i \in \theta\} ,$$

was computed. The time of this operation was measured 20 times and averaged such as to accurately represents the average computational time of a PLU. The choice of this measure was constrained by the available outputs of `FastCodeML`. While this measure is

representative of the average likelihood evaluation cost, it does not differentiate the two previously described scenarios of likelihood evaluation: the one induced by parameters of the evolutionary model (FLEs) and the one induced by parameters representing branch length (PLUs).

Indeed, for this model the parameters are $\theta = (\kappa, p_0, p_1, w_0, w_2, t_1, .., t_{2S-1})$ with the first five parameters being related to the evolutionary model. These five parameters require FLEs, therefore inducing a base computational cost independent of the number of taxa S. This irreducible cost will thus have more impact on the performance gain of data having few taxa.

Constrained partial likelihood update

The first step is to assess the difference in computational time of both implementations when using similar task granularity and, with subtree compression disabled. The average computational time per likelihood evaluation measured for `FastCodeML` and the constrained version of `HOGAN`, limited `HOGAN`, are illustrated in figure 2.9a) and were of similar magnitude for both implementation. However, while the computational time of `FastCodeML` seemed to grow steadily with the number of codons, the one of limited `HOGAN` exhibited a small plateau of performance for small sequences.

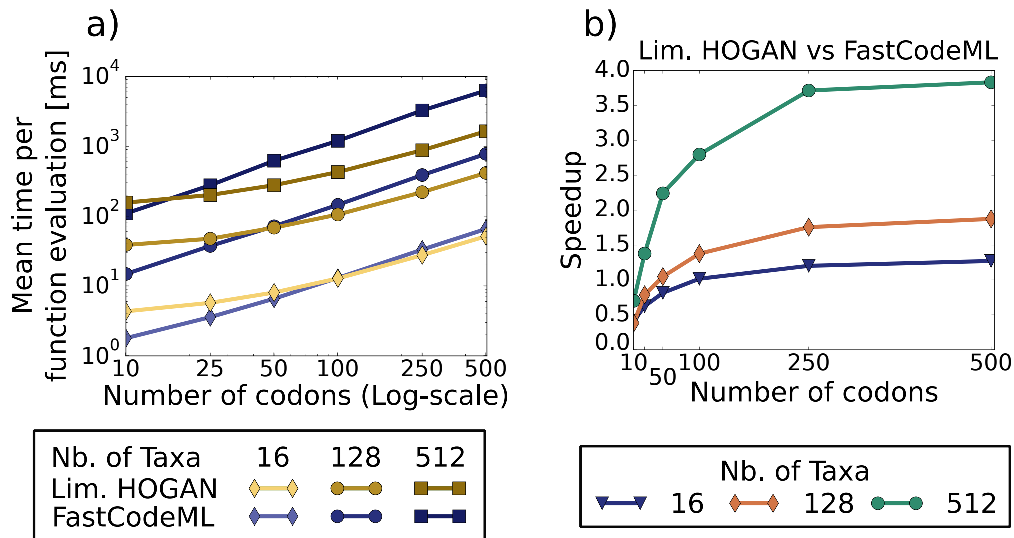


Figure 2.9.: Illustration of `HOGAN` under a similar DAG setting than `FastCodeML` (see Fig. 2.8). Left figure illustrates the average computational time of PLUs for both softwares(log-log scale). Right figure represents the speedup of the limited `HOGAN` when compared to `FastCodeML`.

This behaviour appeared more clearly when considering the computational time speed-up that limited `HOGAN` had over `FastCodeML`. Indeed, as shown in figure 2.9b), limited `HOGAN` exhibited a slowdown on small sequences independently on the size of the tree. This slowdown is explained by the inherent cost induced by the model-generic approach

of `HOGAN`. The management of the DAG and its traversals comes with a computational cost that is not present in `FastCodeML` where the model is *hard coded*. Therefore, the overhead is significant on data instance requiring only few computations per nodes.

As expected, this trend reversed as the number of codons increased. Indeed, the overhead became less significant as the amount of computation per node increased. Moreover this effect was coupled with the improvements brought by compacted CPVs matrices. Indeed, as the sequences size grew, limited `HOGAN` started to exhibit an increase in performance coming from the more efficient matrix-matrix multiplication. Given that this improvement is highly dependent on the cache memory and thus limited by its size, the observed performance increase stabilized as the size of the sequences reached 500 codons.

Moreover, the increase in speedup of limited `HOGAN` was not restricted to the growth of the number of codons. Indeed, it also increased with the size of the tree. This improvement is explained by the ordering of the CPVs computations. Indeed, `FastCodeML` computes the likelihood using a strategy apparent to a post-order breadth-first traversal of the phylogenetic tree, while limited `HOGAN` uses a post-order depth-first traversal. In other words, `FastCodeML` first computes all the leaf nodes and then propagate the results level by level until the root is reached, while limited `HOGAN` computes the CPVs of a node as soon as its dependencies are computed. This last approach enables a better reuse of freshly computed CPVs and thus reduces the overhead caused by data fetching.

In conclusion, limited `HOGAN` exhibited downto twice slower likelihood evaluation times than `FastCodeML` on data having small codon sequences and trees. However, for the large data for which `HOGAN` as been designed, limited `HOGAN` computed the likelihood evaluations upto four times faster than `FastCodeML`. More importantly, this observed performance gain scaled with the size of the tree.

Unconstrained partial likelihood update

In a second step, the full potential of `HOGAN` was analysed by removing the task granularity constraint. The theoretical average performance gain coming from this improvement can be estimated using the bounded average tree height defined in equations (2.8) and (2.9). Indeed, considering that a constrained PLU computes the $2S-1$ CPVs forming the phylogenetic tree, the approximate speedup, $Sp(S)$, is defined as

$$\frac{2S-1}{\bar{h}_w} \leq Sp(S) \leq \frac{2S-1}{\bar{h}_b}.$$

Considering that $h_w \gg 1$ and $h_b \gg 1$ and thus $S \gg 1$, the previous equation can be simplified, using equations (2.10) and (2.11), as

$$\frac{2S-1}{S/2} \approx 4 \leq Sp(S) \leq \frac{2S-1}{\log_2(S)-1} \approx \frac{2S}{\log_2(S)}. \tag{2.12}$$

The speedup of `HOGAN` when compared to limited `HOGAN` illustrated in figure 2.10a) corroborates these theoretical bounds, that are highlighted by a yellow gradient in the

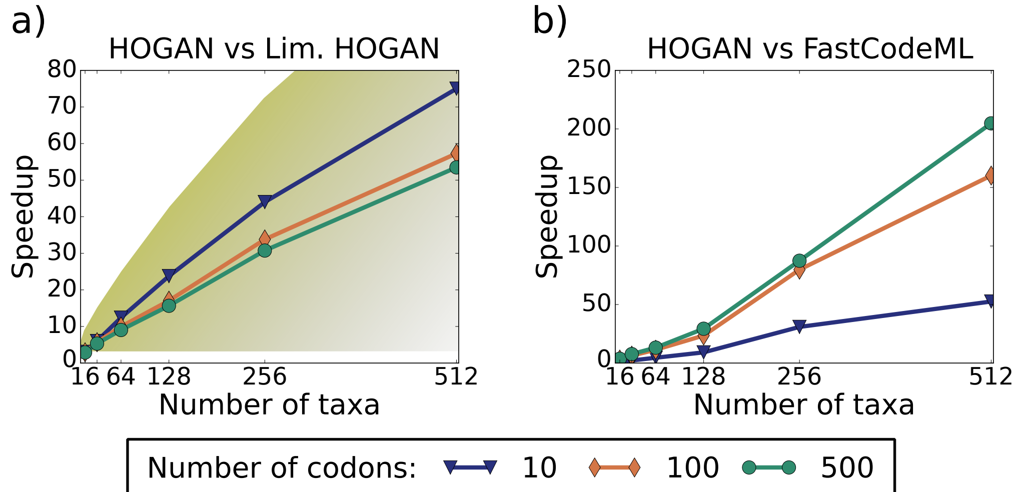


Figure 2.10.: Speedup of `HOGAN` when PLUs are used on the finer DAG representation (as illustrated on Fig. 2.6). Left figure shows the *relative* speedup when compared to `HOGAN` under the same setting as `FastCodeML` (limited `HOGAN`). The yellow gradient expresses the bounded speedup defined by equation (2.12). Right figure shows the *absolute* speedup of `HOGAN` when compared to `FastCodeML`.

figure. Indeed, as shown in figure 2.11, the phylogenetic trees simulated for this experiment were in between the worst and best cases previously illustrated in figure 2.7.

Only phylogenies having 16 taxa showed performance gain lower than the theoretical bounds. However, these results were expected for two reasons : first, in this case $S \not\gg 1$ and second, as discussed previously, the measures of computational time represented the averaged times of likelihood evaluations induced by a single parameter change. Under this circumstance, data instances having small trees, and thus few branches, were strongly impacted by the FLE induced by changes in the evolutionary model parameters.

Although these measures followed the theoretical expectations regarding the tree size, a more surprising effect appeared for data instance having small sequences. Indeed, PLUs further benefited these data that were impacted by the overhead coming from DAG management and traversals in the previous experiments. Given that only parts of the tree are traversed and recomputed when using PLUs, the amount of such operations on the DAG is reduced and thus is their incumbent overhead.

The full potential of `HOGAN` when compared to `FastCodeML` is however more apparent when both methods are directly compared. Indeed, as shown in figure 2.10b), `HOGAN` proved to be as much as 200 times faster than `FastCodeML` when evaluating likelihoods using the large data. Furthermore, these promising performance gains, explained by the combined effect of memory reuse and PLU based on directed task graph, scaled with the tree and sequences sizes.

Finally, the relative speedup of `HOGAN` when using subtree compression is illustrated figure 2.12a). The observed gains from subtree compression are of similar amplitude than

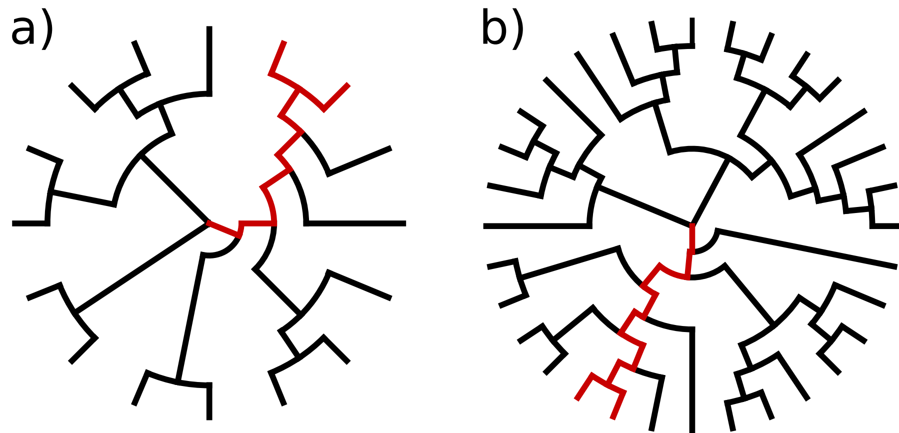


Figure 2.11.: Example of 16 and 32 taxa phylogenetic trees simulated for the dataset. Left figure represents the 16 taxa tree, while right figure show the 32 taxa tree. The deepest lineages are identitified in red and gives maximal tree heights of $h_{16T} = 6$ and $h_{32T} = 8$.

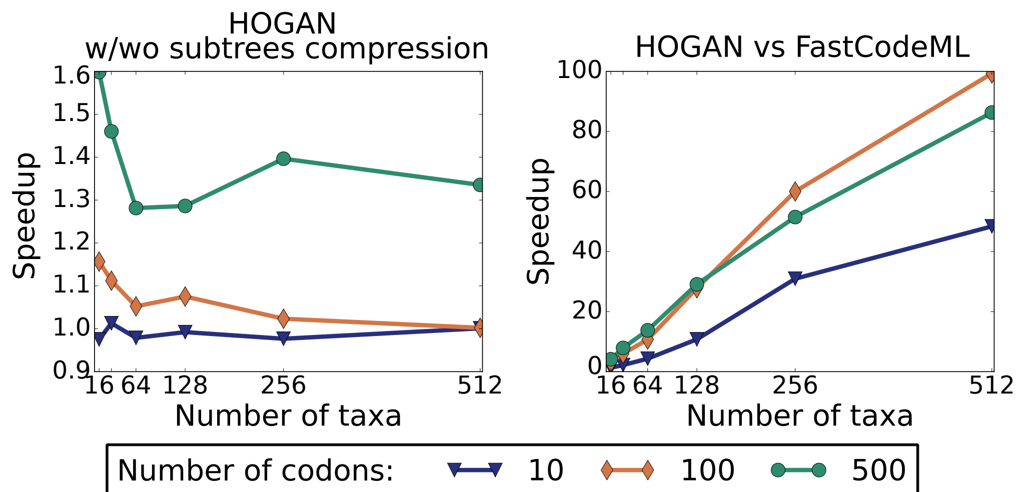


Figure 2.12.: Speedup of `HOGAN` when using subtree compression. Left figure shows the relative speedup of `HOGAN` when enabling subtree compression. Right figure illustrates the absolute speedup of `HOGAN` when compared to `FastCodeML` with both methods having the subtree compression enabled.

the ones documented for `FastCodeML` [127]. More importantly, the absolute speedup of `HOGAN` compared to `FastCodeML`, with both of them having the subtree compression enabled, is represented in figure 2.12b) and confirms the trends observed in the previous experiments.

However, while the speedup on data having 10 codons is nearly identical than in the uncompressed case, data having 100 and 500 codons exhibited smaller speedups. This decrease in performance gain is explained by the amount of compression exploitable in the dataset. Indeed, the data having 10 codons were nearly uncompressed, while data having 100 and 500 codons were compressed from 40% to 60% of their initial size. Large data were thus less benefiting than in the previous experiments from the improvements based on compacted CPVs matrices. In addition of the varying rates of compression, unbalanced compressions over different nodes of the DAG could well explain why data instance having 100 codons outperformed the ones having 500 codons.

2.4. Parallel likelihood evaluations

In addition of being well adapted for PLUs, directed acyclic graphs are widely used for parallel computations. Indeed, the parallel processing of a program requires two fundamental steps. Firstly, the program must be partitioned in several tasks that may express data interdependencies. As described previously, this partitioning is conveniently represented using directed task graphs in which tasks are identified as vertex and dependencies as directed edges. Based on such graphs, the set of tasks being potentially eligible for computation $\mathcal{C}$ is readily identified by the unprocessed tasks v_i having all their parent tasks $\nu(v_i)$ already computed. Thus, by containing the set of tasks having their dependencies fulfilled, this set $\mathcal{C}$ identifies a critical information for parallel processing: the set of tasks having the potential to be concurrently executed.

This set $\mathcal{C}$ evolves differently in function of the task schedule chosen during the DAG traversal. While this choice has nearly no impact on a sequential execution, it plays a fundamental role in parallel computations. Indeed, it defines the order in which dependencies are fulfilled and thus it also indirectly defines the amount of concurrent tasks exploitable $|\mathcal{C}|$. For that reason, the second key step in parallel computations is to determine the scheduling of tasks that minimizes the completion time of the parallel program executed on a defined number of processors n_p.

This scheduling problem exists in two forms: *dynamic* or *static*. Dynamic scheduling is usually applied when no prior information, such as the computational cost of nodes or the amount of communications between them, are known. Under this circumstance, the scheduling is elaborated during the execution of the program. Whenever a processor finishes to process a task, two additional operations are executed: the update of the set of eligible tasks $\mathcal{C}$ and the selection of the next task from $\mathcal{C}$ to compute. To improve the scheduling efficiency, multiple strategies exists for this last step ranging from simple FIFO-like approach to more elaborated approach based on the DAG longest path, more commonly called *critical path* [73]. These elaborated strategies, however, come at the cost of supplementary computations. Therefore, the difficulty of dynamic scheduling

comes from the trade-off between minimizing the computational time of the parallel program and minimizing the overhead coming from the scheduling algorithm.

Oppositely, if the directed task graph is weighted using prior information on the tasks computational cost, static scheduling can be applied. Using this prior knowledge, the task scheduling is statically defined before the execution of the parallel program by finding a good solution to the minimization of the total computational cost. However, defining such a schedule is a NP-Complete problem that has been subject to a vast amount of research during the last decades. Therefore, good task scheduling can be obtained using several algorithms such as `CP/MISF` [67] based on the critical path, `Fastest` [75] based on probabilistic heuristics, `Metis` based on graph partitioning [66] or several others [74].

However, while likelihood computational costs could probably be derived for model-specific implementation, such is not the case for the model-generic approach chosen for `HOGAN`. In addition of lacking prior information on the likelihood computational cost, `HOGAN` generates a specific challenge in regard of the task scheduling due to the use of PLUs. Indeed, as detailed in the previous section, this improvement benefits from the fact that only a sub-part of the DAG is recomputed when few parameters of the statistical model are modified. Assuming a statistical method that modifies one parameter at a time, then, there would be as many different sub-DAGs as there are parameters. On parameter-rich and complex models, finding good task scheduling for each sub-DAG associated with a PLU would then reveal itself a daunting task that could even surpass the computational cost of a statistical analysis of the model.

For these reasons, `HOGAN`'s main scheduler is based on a dynamic scheduling strategy and is implemented on shared memory with POSIX threads. The core algorithm (Algo. 13) is based on a shared FIFO queue that stores the set of nodes $\mathcal{C}$. While being simplistic, this approach is subject to several improvements to increase its performance. For instance, instead of locking the whole DAG during the dependencies update following the computation of a task v_p, a lock is queried for each parents of v_p from a finite set of locks $\mathcal{G}$ (Algo. 15). The attribution of these locks is defined by a mapping function $f(\cdot)$ such that

$$\forall i \in \phi(v_p) \rightarrow g_{f(i)} \in \mathcal{G}.$$

This approach reduces the contention on locks while keeping the size of the pool under control in order to reduce the scheduling algorithm overhead.

Other optimizations are employed to take advantage of memory reuse. For example, whenever a node v_p is computed, its owner thread p updates the parents of v_p, $\nu(v_p)$. During this operation, some parents may become eligible for scheduling. The first one encountered is stored in a *thread-local* buffer, while the others are integrated in the shared tasks set $\mathcal{C}$ (Algo. 15). The scheduling algorithm must then define which task must be computed by thread p. Instead of directly querying $\mathcal{C}$, the thread p checks if a task v_c is stored in his local buffer. If such is the case the global set $\mathcal{C}$ is bypassed and v_c is processed (Algo. 14). In addition to taking advantage of memory reuses[1], this optimization reduces the costly access to $\mathcal{C}$ controlled by a global lock.

[1]Nodes v_p and v_c are computed on the same thread.

In addition of this dynamic scheduling algorithm, HOGAN also has an implementation of a static scheduling strategy. The problem caused by the lack of prior knowledge on the computational cost of the tasks decomposing the likelihood evaluation is circumvented by two different approaches that can be combined. First, an hypothetical unit time can be defined for all the tasks, such that

$$\chi_i = 1, \forall v_i \in \mathcal{V} \tag{2.13}$$

with χ_i defining the computational cost of node v_i. This rough approximation may work under the condition that the tasks have similar computational costs. If this is not the case, then the schedule may be of poor quality.

The second approach takes advantage of the fact that the statistical methods employed to analyse the model are iterative. Therefore, the computational cost of each tasks can be approximated by their average observed computational time $\hat{\chi}$, such that

$$\chi_i = \hat{\chi}_i, \forall v_i \in \mathcal{V}. \tag{2.14}$$

However, these approximations $\hat{\chi}$ are solely useful under the condition that the tasks have a constant execution time and, furthermore, are only accurate after several iterations of the statistical methods. To circumvent this last limitation, the first approach, based on unit times, can be employed until the average observed computational times are judged accurate enough to be employed for the second approach.

This combination of both approaches is implemented in HOGAN in conjunction with the static scheduling strategy CP/MISF [67]. This strategy is a variation of the critical path method that schedules the tasks as to prioritize the execution of nodes comprised in the critical path. Using this static scheduling algorithm, HOGAN defines a good schedule for the directed task graph representing a FLE. The same schedule is then applied to the subgraphs representing the PLUs. While such an approach leads to suboptimal schedules for PLUs, it has the benefit to reduce the overhead coming from the execution of the dynamic scheduling algorithm as well as the one caused by the lock acquisition and contention.

Finally, the choice of a shared memory approach in HOGAN for the parallel computation of likelihood was strongly motivated by the straightforward implementation of dynamic scheduling algorithms on such an architecture. Indeed, as discussed previously, threads can easily reuse memory from previously computed tasks. More importantly, each thread shares the DAG structure and data with the others. Therefore no communications are needed and the memory footprint of the program is reduced. This last advantage is particularly relevant when considering statistical analyses on large datasets that occupy several gigabytes in memory. However, in the event that a more scalable approach should be considered for future likelihood implementations, distributed memory approaches could be readily implemented in HOGAN by taking advantage of its modular communication layers.

2.4.1. Experiments on the branch-site model

In this section, the performance of the scheduling algorithms implemented in HOGAN are illustrated on the branch-site model defined in equation (1.8) and previously used to analyse the performance of PLUs. Being in the continuity of this previous analysis, the forthcoming experiments were based on the previously described datasets and settings (Sec. 2.3). Therefore, the measured speedups of parallel likelihood evaluations were defined as the average speedup over all single parameter PLUs ($|\mathcal{A}| = 1$, see Eq. (2.7)).

However, in order to offer a more instructive benchmark, two changes were made on the previously defined dataset and settings. First, the dataset was augmented by two new sequences, having 1000, respectively 2000 codons, for each phylogenetic tree. Indeed, these new data are used to represent more challenging statistical analysis that would motivate the use of parallel computations. Second, measuring only the averaged speedup over all PLUs is not sufficient to fully grasp the scaling limitations inherent to this model. Therefore in addition of the averaged speedup, a second type of measure differentiates the performance gains of PLUs, that are induced by changes in the evolutionary model parameters, from the one induced by changes in branch lengths. This differentiation of the two classes of parameters offers a finer analysis on the impact that PLUs are having on the parallel computations of likelihood.

The performance of HOGAN when computing the likelihood of branch-site model in parallel is analysed in the following section over three different aspects:

- the tasks granularity;
- the scheduling strategy used: static or dynamic;
- the use, or not, of subtree compression.

For that matter, the default settings used for the upcoming experiments is defined by a fine task granularity, subtree compression disabled and the use of a dynamic scheduling. Each of the experiment aims, by changing one of these setting, to assess the impact of each of the three aspects.

Granularity

In order to measure the impact of the task granularity, two different directed task graphs representing the likelihood were used. The first one represented a fine task granularity that varies in function of the total number of processors P. Indeed, molecular sequences of size N were divided in p subsets of positions of size $l \propto N/P$ that were each used to form a sub-DAG as defined in section 2.2.3). The second directed task graph identified the original likelihood definition where a CPV node was charged with the computation of all the sequence positions, and thus represented a coarse granularity.

The performance gain observed for the coarse granularity case is illustrated in figure 2.13b). The poor scaling of the parallel computation of the likelihood under this setting is readily explainable by the lack of tasks having the potential to be executed in parallel. Indeed, when the granularity was directly adapted to the number of processors,

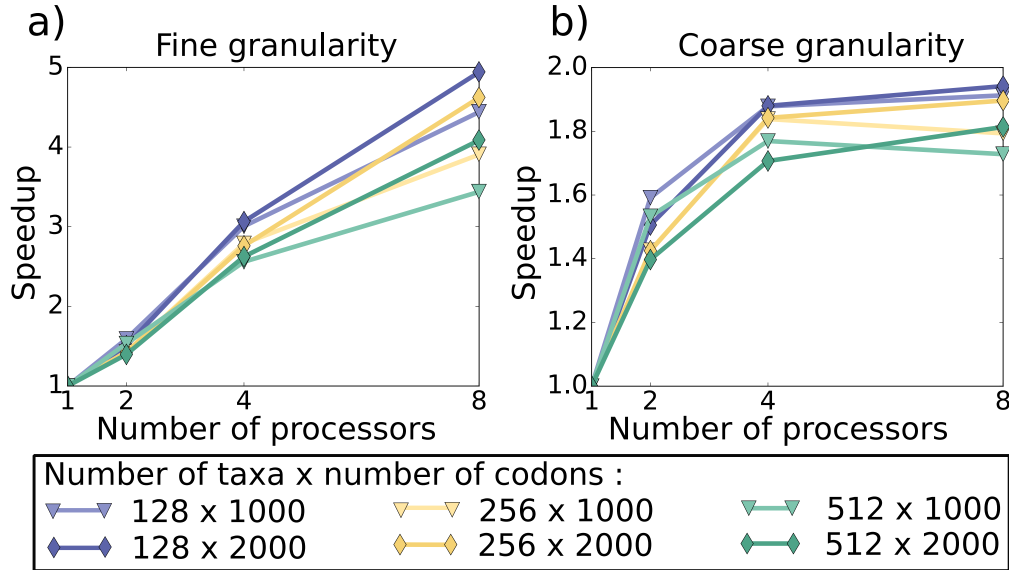


Figure 2.13.: Averaged speedup over PLUs of HOGAN when parallelizing likelihood evaluations with different granularities. Left figure shows the speedup with a fine granularity for leaf and CPV nodes (e.g Fig. 2.4), while right figure shows identical measures with a coarse granularity (e.g Fig. 2.3).

the speedup was scaling more appropriately with the number of processors, as shown in figure 2.13a).

While exhibiting a better scaling, these last measures were still far from optimal. Indeed, using 8 processors only sped up the execution of the likelihood evaluation from a threefold to a fivefold factor. The worst speedup was identified for the data instance having the largest tree and the smallest sequences, while the best one was represented by the opposite data instance having the smallest tree and the largest sequences. Two trends were thus identified: the speedup increased with the size of the sequences and decreased with the size of the tree.

The first trend is explained by the increase in computational cost which is directly linked with the sequences size. Indeed, as the sequences are growing, the overhead caused by the scheduling methods is less prevalent. Moreover, the amount l of positions per subset is also growing, leading thus to improved CPVs computations thanks to the compacted CPVs matrices (Eq. (2.2)).

The second trend is better explained by the effects of PLUs. Indeed, PLUs induced by changes in the five parameters of the evolutionary model requires FLEs. Given the large amount of tasks having the potential to be computed in parallel during such evaluations, nearly linear speedups were obtained on this class of parameters, as shown in figure 2.14a). Oppositely, the amount of computations required for the PLUs induced by changes in the branch lengths were far smaller and thus presented a more limited amount of exploitable parallelism.

Indeed, such PLUs are represented by a directed trail in the directed task graph. While

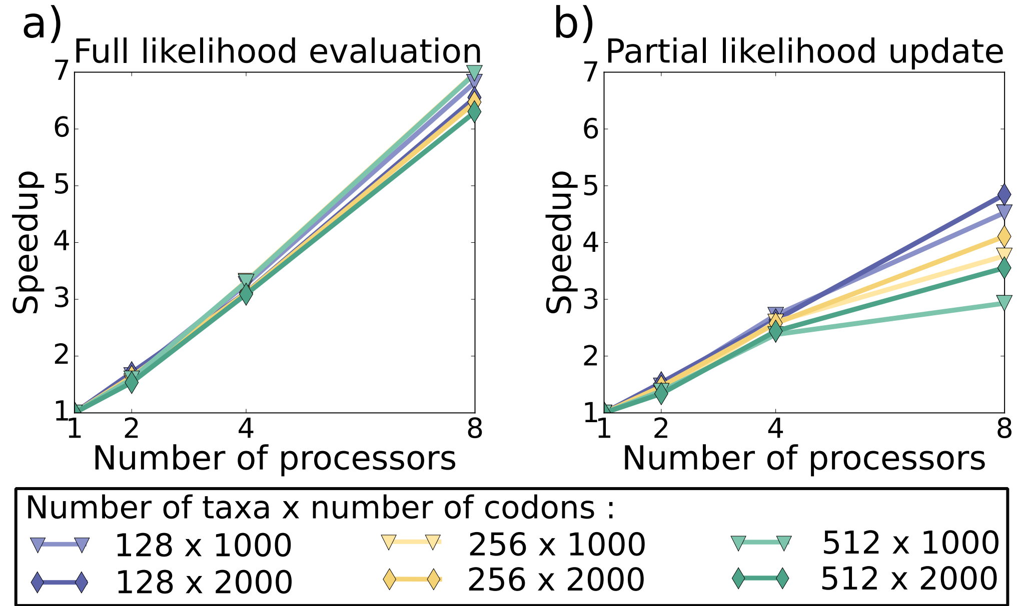


Figure 2.14.: Speedup of HOGAN when parallelizing FLEs, that are required for the evolutionary model parameters (left figure), and PLUs, occuring when a single branch length is changed (right figure).

a finer granularity for CPV tasks helps to parallelize this inherently sequential structure, it does not offer any improvements in regard of the starting branch-matrix task of the trail, as well as the ending root task (Fig. 2.6). While being already problematic, this lack of potential parallelism also leads to an increased level of thread contention that grows proportionally with the length of the trail. Therefore the speedups obtained on this class of parameters were worse than the ones obtained for the other class and, as illustrated in figure 2.14b), were decreasing with the size of the tree.

Additionally to this first decrease in speedup caused by an increase of tree size, a second combined effect was due to the induced increase in branch length parameters. Indeed, the averaged speedup over all PLUs is defined as

$$\frac{5 \times Sp_{Evo} + 2S \times Sp_{BL}}{5 + 2S} \tag{2.15}$$

with Sp_{Evo}, and Sp_{BL}, defining the average speedups for PLUs induced by the first, respectively second, class of parameters. Therefore, the nearly linear speedups, Sp_{Evo}, obtained for the five evolutionary parameters are weighing significantly on the averaged speedup solely for small trees. However as soon as $S \gg 5$, the gains coming from Sp_{Evo} only faintly impact the averaged speedup which is then mainly representing the speedup of the second class of parameters, Sp_{BL}.

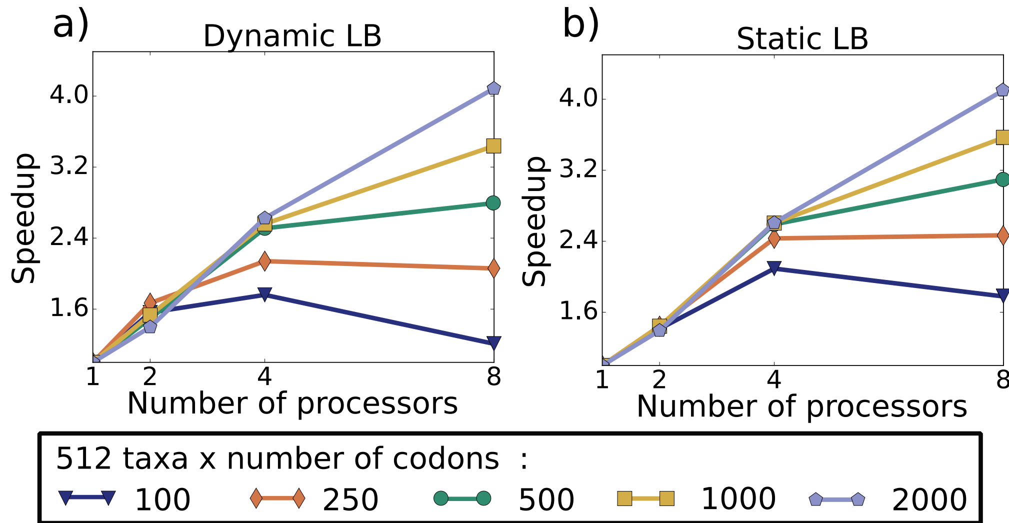


Figure 2.15.: Averaged speedup over PLUs of HOGAN when parallelizing likelihood evaluations with dynamic (left figure) and static (right figure) load balancing (LB).

Scheduling strategies

In order to identify the potential role of the scheduling strategy on the previously observed trends, the dynamic and static scheduling implementations were compared. As shown in figure 2.15, this comparison revealed that the overhead caused by the dynamic scheduling (Fig. 2.15a)) was mostly impacting the data having few codons. Indeed, the static scheduling (Fig. 2.15b)) was showing slightly improved speedups for these data. However as the amount of codons in the sequences grew, the difference in performance gain between both strategies reduced until becoming non-existent for data having 2000 codons.

Therefore, while the static scheduling seems to slightly reduce the scheduling overhead and thus improve the performance, this strategy does not solve the limitation caused by branch length induced PLUs that was identified in the previous section. Indeed, static scheduling does not enable parallel computation of branch-matrix and root tasks. Moreover, as previously discussed, the unique static schedule of the tasks defined for the whole directed task graphs cannot represent an optimal schedule for each different sub-DAG associated with a PLU.

Subtree compression

The previous experiments were conducted without using subtree compression as to accurately assess the impact of the sequences size on the behaviour of the likelihood parallel processing. Therefore, the impact that this form of compression has on parallel likelihood computation is now analysed, starting with the relative speedup. This speedup was obtained by using subtree compression for both the sequential and parallel computations.

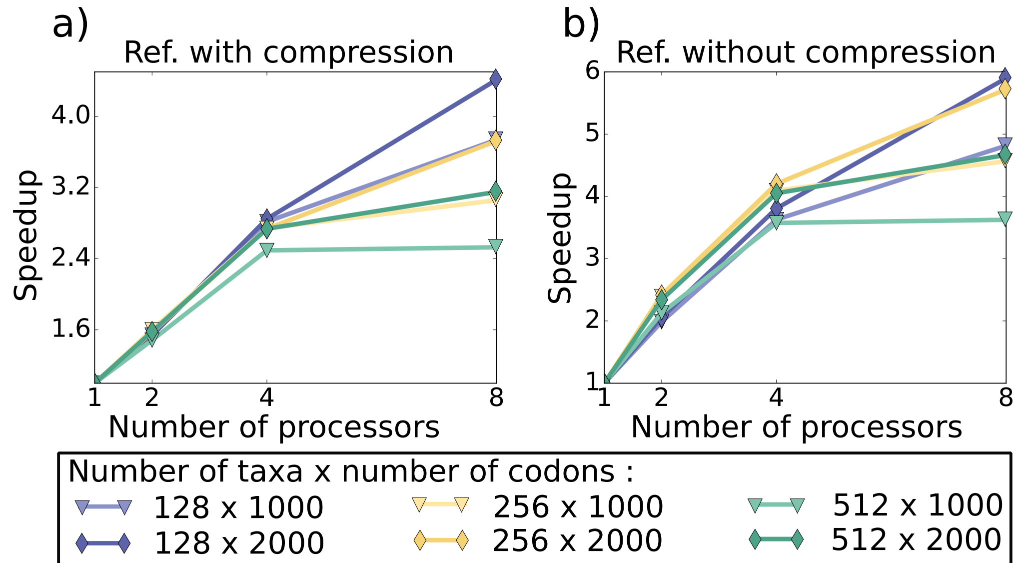


Figure 2.16.: Averaged speedup over PLUs of `HOGAN` when parallelizing likelihood evaluations with subtree compression. On the left figure, respectively right figure, the subtree compression is enabled, respectively disabled, for the reference sequential implementation.

As shown in figure 2.16a), this relative speedup exhibited the same trends as the previous experiments. However, given that the CPVs were compressed, less computations where performed on each task and thus the speedups were slightly worse than the ones observed without subtree compression (Fig. 2.13a)). This observation confirmed that the amount of computations per CPV task, which is linked with the amount of codons, played a important role on the scaling of the parallel processing of this likelihood.

While the subtree compression led to a slight loss of relative speedup, its use was still beneficial in regards of the overall performance. Indeed, the absolute speedup, obtained by comparing sequential computations without compression to parallel computation with compression, still represented the best observed speedup. Indeed, as illustrated in figure 2.16b), the absolute speedup exceeded the ones obtained when no subtree compression were used (Fig. 2.13a)).

2.5. Summary

In this chapter, the choice of DAGs as a representation for statistical models and likelihood functions in HOGAN is justified. More precisely, the advantages of using a directed task graph over a graphical model representation are detailed. Then, two approaches that enable efficient likelihood evaluations are described from a model-generic point of view and illustrated on the branch-site model detecting positive selection.

Model-generic

The two different model-generic methods, made possible by the likelihood representation as directed task graph, are:

- The formalization of partial likelihood updates (PLUs) that enable the reuse of previously computed partial results.
- The parallel computation of likelihood evaluation using dynamic and static load balancing strategies.
  - Static load balancing strategies are made possible by using estimations of the tasks cost:
    - by assuming a unit task cost;
    - by using the observed average execution time of tasks at runtime.

Model-specific

The implementation in HOGAN of the central phylogenetic tree based likelihood is thoroughly described and theoretical limits for the PLUs induced performance gains on this type of models are derived as a function of the tree structure.

This implementation is validated on the branch-site model detecting positive selection and compared with the state-of-the-art implementation FastCodeML on several synthetic datasets.

- Without PLUs and parallel likelihood evaluations, HOGAN implementation outperforms significantly FastCodeML on large datasets.
- With PLUs activated, HOGAN reduces the complexity of likelihood evaluations by upto a factor $2S/\log_2(S)$ with S being the number of taxa.
- When parallel likelihood evaluations are added on top of PLUs, HOGAN further accelerates likelihood evaluations by upto a factor 4 with 8 processors.

3. Maximum likelihood estimation

Maximum likelihood estimation is a method for choosing estimators of the parameters of a statistical model given data without considering prior distributions. This method chooses as the estimate of parameters $\theta \in \Theta$ a value that maximizes the goodness of fit of the statistical model given the observed data X, as defined by

$$\hat{\theta}_{mle} \subseteq \left\{ \arg\max_{\theta \in \Theta} f(X|\theta) \right\}. \tag{3.1}$$

In addition of serving the purpose of estimating the model parameters, ML estimation plays a central role in the comparison of hypotheses by means of statistical tests such as likelihood-ratio test. In evolutionary biology, these tests are frequently employed to determine if an elaborated evolutionary hypothesis better explains the observed data than a simpler one[1]. For example, the relevance of codon models identifying the positive selection is assessed by confronting their fitting quality to the one of their null hypothesis that represents a special case of the models where positive selection cannot be expressed. However for such complex models, having potentially a large amount of parameters, the estimation of the ML is a costly operation.

Indeed, for such models, the maximization of $f(X|\theta)$ is done through the use of optimization methods that must explore their highly dimensional and complex parameter spaces. Thankfully, whenever the parameters θ are continuous, well-defined continuous optimizations methods [91], such as the gradient descent or *Newton* method, can be employed for that matter. These methods avoid a blind exploration of the parameter space by using the gradient of the likelihood function $\nabla f(X|\theta)$ to define the steepest direction toward a maxima. However, given that the analytic form of $\nabla f(X|\theta)$ is generally unknown, the computation of the direction, required for each optimization step, comes at the expensive cost of a computational approximation of the gradient.

This chapter begins by shortly introducing the concepts of the continuous optimization methods. Following this introduction, two generic strategies accelerating the computations of an optimization step are presented. The first strategy takes advantage of the DAG representation of a likelihood to define an optimal computational scheme for the approximation of the gradient. Using this optimal scheme, the second strategy proposes a scheduling and load balancing algorithm for the parallel computation of this expensive approximation. Finally, as in the previous chapter, the behaviour of both methods are analysed on the branch-site model defined in equation (1.8).

[1] The simpler hypothesis must be nested in the elaborated one.

3.1. Basic concepts of continuous optimization

This section gives a brief overview, largely inspired from the unavoidable book *Numerical Optimization* [91], of the continuous optimization algorithms used in HOGAN. In this context, these algorithms are applied to the maximization of the likelihood, as defined by equation (3.1). However, in a more general context, these algorithms optimise an *objective function* $f(x)$ as

$$\min_x f(x) = -\max_x (-f(x)) \tag{3.2}$$

with $x \in \mathbb{R}^d$ defining a real vector and $f : \mathbb{R}^d \to \mathbb{R}$ defining a function having continuous derivatives of all orders everywhere in its domain.

A *local solution* of such minimization problem is defined as a stationary point $x^\star$ guaranteeing that $\nabla f(x^\star) = 0$ and that $\nabla^2 f(x^\star)$ is *positive semidefinite*. More intuitively put, these conditions verify that there is no other points in the immediate neighbourhood $\mathcal{N}$ of $x^\star$ such that $f(x) < f(x^\star)\ \forall\ x \in \mathcal{N}$. These solutions are thus identified as local given that $x^\star$ only guarantees their optimality with respect to their immediate neighbourhood. By extension, a *global solution* guarantees, over the whole function domain, that there are no other points such that $f(x) < f(x^\star)\ \forall\ x \in \mathbb{R}^d$.

To reach a solution, optimization algorithms iteratively generate a sequence of points $\{x_k\}_{k=0}^{\infty}$ ensuring that the function value progresses toward the minima as k increases such that $f(x_{k+1}) < f(x_k)$. This sequence starts with the point x_0 that is either arbitrarily chosen or supplied by the user whenever a reasonable estimate of $x^\star$ is available. It then terminates when no more progress can be made, with respect to a predefined accuracy tolerance, and thus its last element defines the point $x^\star$.

The strategies employed to generate the next point x_{k+1} are generally using information about the function f at the current point x_k, as well as the previous points in the sequence $(x_0, .., x_{k-1})$. The algorithms used in HOGAN are based on the *line search* strategy that searches, along a direction p_k, a new point that would lower the objective function value, as

$$x_{k+1} = x_k + \alpha_k p_k$$

The efficiency of this strategy depends thus on the definition of the step length α_k and the search direction p_k. The choice of both of these values presents a trade-off between improving the convergence rate of the algorithm toward a solution and maintaining a low computational complexity. For example, while the step length α_k could be optimally defined by solving the minimization problem defined as

$$\min_{\alpha_k > 0} f(x_k + \alpha_k p_k),$$

cheaper approaches are preferred. Indeed, typical line search algorithms approximate its value by trying iteratively values for α_k until a set of conditions, such as the *Wolfe* or *Goldstein* conditions, are met [91].

For its part, the search direction p_k is generally defined as

$$p_k = -B_k^{-1} \nabla f(x_k)$$

with B_k representing a symmetric invertible matrix. Depending on the choice of this matrix, the strategy can either be the simple *gradient descent* method with $B_k = I$ and I being the identity matrix, or the more evolved Newton method that requires B_k to be the exact Hessian of the objective function, $\nabla^2 f(x)$. While this last method highly surpasses the convergence rate of the steepest descent method, it also requires the expensive computations of the Hessian and its inversion.

By using an approximation of the Hessian, *quasi-Newton* methods are able to benefit from a convergence rate competitive with the Newton's method without suffering from its expensive cost. Indeed, in these methods, the B_k matrix is a lower-rank approximation of the Hessian updated at each iteration in function of the current point and information collected during previous iterations. Based on this approach, the well known *Broyden–Fletcher–Goldfarb–Shanno* algorithm (BFGS) defines an updating scheme of B_k such that its most expensive computational costs are the gradient computations and the matrix-matrix multiplications required for the update of B_k.

However, for high-dimensional models, the size of B_k becomes significant as well as the operations required to keep it updated. The limited-memory BFGS algorithm (LBFGS) reduces this overhead by solely using the m most recent information about the function curvature to approximate the Hessian at each step. This approximation has the advantage of being *implicitly* stored as m vectors of dimension n and thus, depending on the number m supplied by the user, significantly reduces the amount of memory and computation required to compute B_k.

Both algorithms are used in `HOGAN`. The first one, BFGS, is directly implemented in the framework and is used in its standalone version. However, the LBFGS algorithm is more efficient on high-dimensional models and the robust implementation from the `NLopt` library [63] is preferred on such models.

3.1.1. Computing the derivatives

The presented algorithms define an efficient setting, with a controlled computational cost, for the optimization of a smooth function f. This setting however largely depends on the availability of the gradient $\nabla f(x)$ of this function. Most of the time, the analytic form of $\nabla f(x)$ is unknown and must thus be obtained using computational approaches. Two of them can potentially fulfil the requirements of `HOGAN`: the *automatic differentiation* [90] and the *method of finite differences*, also known as *numerical differentiation* [91].

Automatic differentiation

Automatic differentiation (AD) decomposes the function f in elementary arithmetic operations in order to apply the *chain rule*. Using this approach, derivative with an accuracy up to the machine precision can be obtained either by using forward or reverse accumulation. The forward variant traverses the chain rule from inside to outside and obtains the derivative at a cost proportional to the dimension of x.

As the name of the second variant suggests, the chain rule is traversed in the opposite direction, from outside to inside, and calculates the derivative at a small constant mul-

tiple, usually between 5 and 30, of the cost of the original function evaluation. While presenting an impressively low cost by being independent of the dimension of x, this variant suffers from two limitations for function f representing large computations: its memory footprint grows proportionally with the amount of elementary operations in the algorithm and more importantly its joint use with functions implemented on parallel architecture is difficult [48].

These memory issues can be addressed by using advanced strategies known as *checkpointing* that relax the memory requirements at the cost of additional computations [129, 23]. On the other hand, integration of the reverse accumulation on parallel architectures remains an open question [89, 126] that grow in complexity when *checkpointing* is taken into consideration [55]. To our knowledge, successful use of parallel architecture has been limited to proofs of concept [110] or expert software [69].

Method of finite differences

Compared to automatic differentiation, the method of finite differences is straightforward to implement. Indeed, the derivatives are simply approximated as

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x+\epsilon e_i) - f(x)}{\epsilon}, \text{ or} \tag{3.3}$$

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x+\epsilon e_i) - f(x-\epsilon e_i)}{2\epsilon}, \tag{3.4}$$

with ϵ being a small positive scalar and e_i the ith unit vector. The first equation defines the *forward-difference* approximation and has a cost proportional with the dimension of x, similarly as the forward accumulation variant of AD. The second one defines the *central-difference* approximation and, comparatively to the previous equation, increases the derivative accuracy from $\mathcal{O}(\epsilon)$ to $\mathcal{O}(\epsilon^2)$ for twice its computational cost.

Therefore, the derivative accuracy is controlled by the choice of ϵ, which is in turn defined by the accuracy of the machine. Indeed, the accuracy of floating-points operations is bounded by the *unit roundoff* $\mathbf{u}$ that represents the relative error on arithmetic operations between two variables. A good choice for ϵ can thus be derived from the propagation of errors during the derivative computation and is given as $\epsilon = \sqrt{\mathbf{u}}$.

Derivatives computation in `HOGAN`

Automatic differentiation was not chosen for `HOGAN` for several reasons. The current AD libraries[2] [90] interact with existing codes through two means: either by parsing the existing code and generating the required functions for the derivatives computation, or by furnishing specialised operators that must overload the elementary arithmetic operators. The first approach is crippling for a model-generic framework, while the second is code intrusive, disables the use of external libraries and would require to be well integrated in order to be transparent to model developers.

[2] List of these libraries: `http://www.autodiff.org`

A full integration in the framework of AD methods could thus be envisioned. However, such integration would require large development and research efforts, more so in the context of a framework built for parallel computation and large dataset. This last criterion could be problematic with respect to the already significant memory requirement of the reverse accumulation variant. Therefore, the only, yet notable, advantage would be the increased derivative accuracy brought by the forward accumulation.

The current implementation of `HOGAN` is, for these reasons, based on the method of finite differences. In addition of its straightforward implementation, this method directly benefits from the reduced cost of the PLUs given that it computes the partial derivatives by modifying one by one the elements of x (Eq. (3.3)). Moreover, as shown in the next sections, non intrusive and generic methods that reduce the computation cost of this expensive step can be designed using the DAG representation of likelihoods.

3.2. Finite difference approximation of likelihoods represented as DAGs

In `HOGAN`, the ML estimate (Eq. 3.1) of a smooth likelihood function, $f(X|\theta)$, is obtained through the use of the continuous optimization methods defined previously. For that matter, the gradient of the likelihood function, defined as

$$\nabla f(X|\theta) = \frac{\partial f}{\partial \theta_1} e_1 + ... + \frac{\partial f}{\partial \theta_d} e_d,$$

is computed at each step of the optimization using the finite differences method defined by equations (3.3) or (3.4)[3].

The major caveat of this method is the expensive computational cost caused by the evaluations of the $m+1$ ($2m$ for Eq. (3.4)) likelihood functions that are required for the approximation of the gradient. Indeed, the likelihood has first to be evaluated with the current values of the vector of parameters θ. This computation is then followed by m subsequent evaluations representing the ϵ perturbation applied to the each element of θ. The operations required for the gradient approximation can therefore be represented as the following sequence:

$$\mathcal{S} = \Big(f(X|\theta), f(X|\theta + \epsilon e_1), ..., f(X|\theta + \epsilon e_m)\Big). \tag{3.5}$$

This expensive sequence of evaluations can be drastically reduced thanks to the DAG representation of likelihood used in `HOGAN`. Indeed, apart for the first evaluation in the sequence that requires a FLE, all the subsequent ones are PLUs induced by the application of perturbations ϵe_i and thus can benefit from the scheme defined in section 2.3 that reduce the computational cost of such computations.

The use of PLU can be enabled by maintaining a copy in memory of the state of the DAG and its partial results as a checkpoint after the evaluation of $f(X|\theta)$. Indeed, this checkpoint could be employed to reset the state of the DAG prior to each perturbation

[3]The upcoming explanation is based on the first formulation, however it also holds for the second one.

evaluation ϵe_i that would then solely require a PLU defined by the set of nodes $\Psi_{\{\theta_i\}}$. While this approach can drastically increase the efficiency of the gradients evaluation, it may reveal infeasible in a memory-limited context due to the expensive memory cost of maintaining a second copy of the data dependencies.

In order to still benefit from PLUs without increasing the memory footprint, the computations sequence $\mathcal{S}$ can be reformulated by assuming an arbitrary order of the perturbations ϵe_i. These function evaluations can then be defined as the sequence

$$\mathcal{S} = \Big(f(X|\theta^{(0)}), f(X|\theta^{(1)}), ..., f(X|\theta^{(i)}), ..., f(X|\theta^{(d)})\Big) \tag{3.6}$$

with

$$\begin{aligned} \theta^{(0)} &= \theta, \\ \theta^{(1)} &= \theta^{(0)} + \epsilon e_1, \\ \theta^{(2)} &= \theta^{(1)} - \epsilon e_1 + \epsilon e_2, \\ \theta^{(i)} &= \theta^{(i-1)} - \epsilon e_{i-1} + \epsilon e_i, \\ \theta^{(d)} &= \theta^{(d-1)} - \epsilon e_{d-1} + \epsilon e_d. \end{aligned} \tag{3.7}$$

From this formulation it clearly becomes apparent that, starting from the third likelihood evaluation in the sequence, each evaluation consists in a partial likelihood evaluation that first cancels the effect of the perturbation on the previous element e_{i-1} of θ and then applies the current perturbation to the current element e_i. Therefore, these partial likelihoods are induced by the changes of two parameters: θ_{i-1} and θ_i.

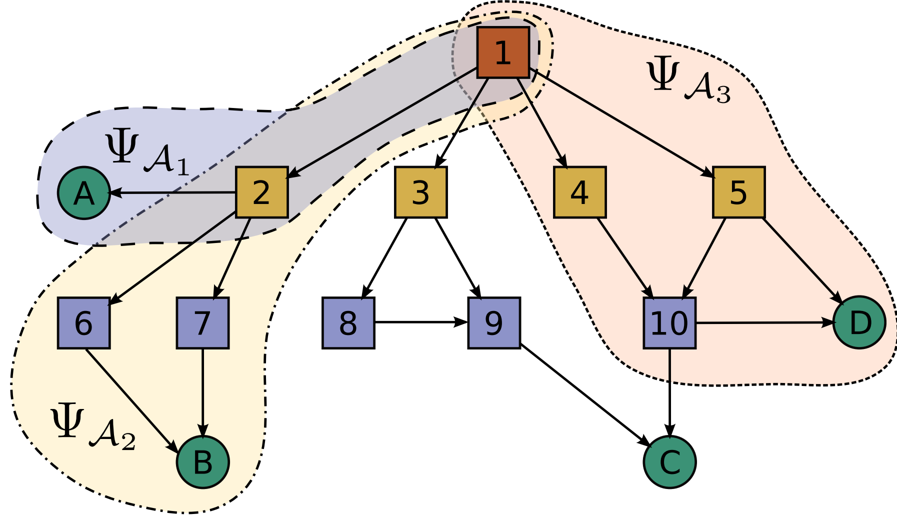


Figure 3.1.: Example of PLUs on a directed task graph. Three tasks sets, $(\Psi_{\mathcal{A}_1}, \Psi_{\mathcal{A}_2}, \Psi_{\mathcal{A}_3})$ requiring to be recomputed after the change of their respective parameters (A, B, D) are highlighted.

The set of tasks to recompute, $\Psi_{\mathcal{A}}$, during such PLU identified by changes in the set

of parameters $\mathcal{A}$, is thus defined as

$$\Psi_{\mathcal{A}} = \Psi_{\{\theta_{i-1},\theta_i\}} = \Psi_{\{\theta_{i-1}\}} \cup \Psi_{\{\theta_i\}}. \tag{3.8}$$

This crucial operation occurring during the gradient computation is explained using the sets of tasks represented in figure 3.1. For instance, the sets $\Psi_{\mathcal{A}_1} = \{v_1, v_2\}$, or $\Psi_{\mathcal{A}_3} = \{v_1, v_4, v_5, v_{10}\}$, have to be recomputed after a change of parameters A, or respectively D. If, during the gradient approximation, the perturbation of parameter A follows the one of D, then the set of tasks representing the reset of nodes $\Psi_{\mathcal{A}_3}$ combined with the computations $\Psi_{\mathcal{A}_1}$ induced by the last perturbation, gives

$$\Psi_{\mathcal{A}} = \Psi_{\mathcal{A}_3} \cup \Psi_{\mathcal{A}_1} = \{v_1, v_2, v_4, v_5, v_{10}\}.$$

More importantly, the ordering, or permutation, $\mathcal{R}$ of the perturbations $(e_i\epsilon)$ identified in equations (3.7) such that

$$(e_1, e_2, .., e_i, .., m) \to \mathcal{R} = (1, 2, .., i, .., m),$$

is not constrained by any means. Indeed, the only mandatory steps to fulfil for the approximation of the gradient is the evaluations of all the likelihood functions contained in the sequence $\mathcal{S}$ regardless of their ordering. Therefore all the possible permutations of the perturbation order guarantee a correct gradient approximation. However, the computational cost induced by different permutations, $\mathcal{R}$, is not identical given that the pairs of parameters $\mathcal{A}$ (Eq. (3.8)) defining the PLUs are enforced by the ordering.

The effects of the permutations become clearly apparent when considering two different perturbation ordering for the parameters (A, B, D) represented in figure 3.1. If the first FLE $f(X|\theta)$ is omitted, then the permutation, (A, B, D), gives the following sequences of tasks to compute,

$$\begin{aligned}
A + \epsilon \to \quad & \Psi_{\mathcal{A}} = \Psi_{\mathcal{A}_1} = \{v_1, v_2\}, \\
A - \epsilon, B + \epsilon \to \quad & \Psi_{\mathcal{A}} = \Psi_{\mathcal{A}_1} \cup \Psi_{\mathcal{A}_2} = \{v_1, v_2, v_6, v_7\}, \\
B - \epsilon, D + \epsilon \to \quad & \Psi_{\mathcal{A}} = \Psi_{\mathcal{A}_2} \cup \Psi_{\mathcal{A}_3} = \{v_1, v_2, v_4, v_5, v_6, v_7, v_{10}\}.
\end{aligned}$$

The total amount of tasks computed for this permutation amounts to 13 and has an identical computational cost when assuming a unit computational costs for each of the tasks.

By applying the same procedure for the permutation, (B, D, A), the following sequences of tasks must be computed,

$$\begin{aligned}
B + \epsilon \to \quad & \Psi_{\mathcal{A}} = \Psi_{\mathcal{A}_2} = \{v_1, v_2, v_6, v_7\}, \\
B - \epsilon, D + \epsilon \to \quad & \Psi_{\mathcal{A}} = \Psi_{\mathcal{A}_2} \cup \Psi_{\mathcal{A}_3} = \{v_1, v_2, v_4, v_5, v_6, v_7, v_{10}\}, \\
D - \epsilon, A + \epsilon \to \quad & \Psi_{\mathcal{A}} = \Psi_{\mathcal{A}_3} \cup \Psi_{\mathcal{A}_1} = \{v_1, v_2, v_4, v_5, v_{10}\}.
\end{aligned}$$

This second permutation has a computational cost of 16 which represents roughly a 20% increase compared to the first permutation.

This example highlights the importance of an adequate ordering of the perturbations. Indeed, if a bad scheduling of the computations required for the gradient approximation can have a noticeable effect on such a simple DAG, then on more complex DAG the performance of the optimization algorithm could well be significantly undermined.

3.3. Perturbations scheduling in gradient approximations

Finding the optimal permutation $\mathcal{R}$ that minimizes the computational cost of the gradient approximation is a daunting task. Indeed, the number of possible permutations is defined as

$$(\mathcal{R})^d_d = \binom{d}{d} = d! \tag{3.9}$$

This amount of permutations becomes thus quickly intractable. For instance, ten parameters have already more than three million possible permutations.

This problem can however be formalised by using the previous observations and features defined for likelihoods represented as DAG. Considering known computational costs χ_s for all tasks $v_s \in \mathcal{V}$ and a permutation $\mathcal{R}$, the computational cost of the sequence expressed in equation (3.6) can be defined as

$$\mathcal{C}(\mathcal{R}) = \sum_{i=0}^{d} \mathcal{C}_i(\mathcal{R}) = \sum_{i=0}^{d} \left(\sum_{v_s \in \Psi_{\mathcal{A}_i}} \chi_s \right) \tag{3.10}$$

with

$$\begin{aligned} i = 0 \quad &\longrightarrow \quad \Psi_{\mathcal{A}_0} \text{ with } \mathcal{A}_0 = \{\theta_.\}\,, \\ i = 1 \quad &\longrightarrow \quad \Psi_{\mathcal{A}_1} \text{ with } \mathcal{A}_1 = \{\theta_{\mathcal{R}_1}\}\,, \\ i \geq 2 \quad &\longrightarrow \quad \Psi_{\mathcal{A}_i} \text{ with } \mathcal{A}_i = \{\theta_{\mathcal{R}_{i-1}}, \theta_{\mathcal{R}_i}\}\,. \end{aligned}$$

The optimal perturbation schedule is determined by the permutation $\mathcal{R}$ that minimizes the computational cost of the full sequence. Given that the first step, $i = 0$ is independent of the permutation $\mathcal{R}$, it can be omitted in the minimization of the cost that is then defined by

$$\min_{\mathcal{R}} \sum_{i=1}^{d} \mathcal{C}_i(\mathcal{R}). \tag{3.11}$$

This definition of the problem relies heavily on the assumption that the computational costs χ_s are known beforehand. While this is not usually the case, more so in a model-generic framework, this issue was already addressed in the previous chapter during the description of the scheduling methods for parallel likelihood evaluations. Therefore, using the same approaches, the costs χ_s can be approximated either by using unit task costs (Eq. (2.13)) or by using the average observed computational time per task (Eq. (2.14)).

Luckily enough, this problem is one of the most widely studied in computational mathematics. Indeed, by considering the likelihood evaluation forming the sequence $\mathcal{S}$ (Eq. (3.5)) as the vertices $\mathcal{V}$ of a graph and the computational cost between a pair of them (Eq (3.10)) as the weighted edges $\{w_i : e_i \in \mathcal{E}\}$, the problem at hand consists in finding the trail that starts from the first vertex, visits only once all the vertices and defines the minimal possible total distance. By renaming *vertex* by *city*, it becomes directly apparent that this is an instance of the *Travelling Salesman Problem* (TSP) aiming "*to find the cheapest way to visit all the cities*" [7].

Since the 1950s, the quest of methodologies enabling to find exact solution for this NP-Hard problem has continued unabated. In 1954, a, then challenging, instance with 54 cities was solved using the *cutting-plane* method. In 1987, exact solutions for instances having few thousands cities were successfully determined using the *branch-and-bound* method. In the early 2000s, using the *domino-parity* algorithm the optimal tour for an instance having 85,900 cities was found.

In parallel of these exact, but expensive, approaches, several heuristic methods aiming to give good solutions in a short amount of time were developed. Through the years, the *Lin-Kernighan* (LK) family of heuristics emerged as one of the most, if not the most efficient approach to find good solution for large TSP instances. Indeed, in 2001, during the 8th DIMACS[4] Implementation Challenge [62] dedicated to the TSP, several heuristics implementation were compared on standardised benchmarks. During this challenge, the LK-based heuristics consistently dominated the other approaches; performance equally attributed to the heuristic efficiency and the implementation details [37, 61].

For these reasons, in order to find a good solution to the problem expressed by equation (3.11), `HOGAN` uses an available implementation[5] of an iterative variant of the LK heuristics, known as `LKH` [57]. This variant uses the *iterated LK* algorithm that intelligently launches several succinct LK searches to improve the solution quality.

3.3.1. Experiments on the branch-site model (cont.)

This section is dedicated to the analysis of the potential gains coming from the scheduling of perturbations during a gradient approximation on the branch-site model defined in equation (1.8). Therefore, the experiments conducted for this analysis were in the same line as the ones of the previous chapter and thus the same datasets and general settings were used.

The previous experiments were considering improvements on a single likelihood evaluation. Given that an existing state-of-the-art implementation was available, assessing the performance of a single likelihood evaluation was rather straightforward. Evaluating the effects of perturbations scheduling on the computational cost of a gradient approximation is however more complex. Indeed, in order to adequately assess the performance gain of different perturbations schedules on the branch-site model, some points have first to be addressed:

1. Can bad or good perturbations schedule be identified for this model ?
2. Does a theoretical bound on the performance gain exist ?
3. What kind of schedule should be used as reference ?

The first question can be answered by considering the previous observations done on the simulated phylogenetic tree of the dataset. Indeed, these trees had a topology that was close to a perfect binary tree (Fig. 2.11). Therefore, this topology forms a good base to study the worst and best-case scenarios of the scheduling of perturbations.

[4]DIMACS stands for "Center for Discrete Mathematics and Theoretical Computer Science".
[5]`http://www.akira.ruc.dk/~keld/research/LKH/`

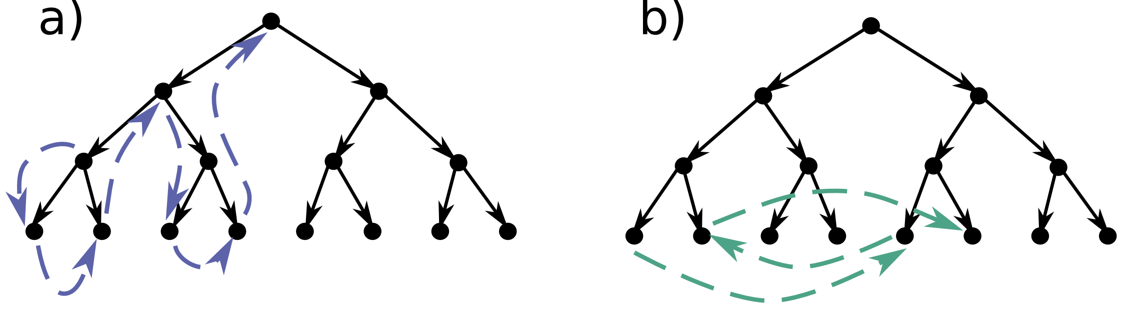


Figure 3.2.: Rough simplification of DAGs representing the tree likelihood; the Matrix task is omitted and the CPV or Leaf tasks are represented conjointly with their Branch-Matrix tasks as a single vertex. The left and right figures illustrate then the best-case, respectively worst-case, scenario for perturbations scheduling.

In the previous experiments of the branch-site model, two different classes of parameters were defined: the ones of the evolutionary model requiring FLEs and the ones of the branch lengths inducing PLUs. Given that the first classe has a fixed number of five parameters that do not induce PLUs upon modifications, the quality of the perturbations schedule can be safely assumed to be mainly impacted by the branch length parameters.

Each of these parameters is directly related with the edges of the tree topology that in turn form the backbone of the DAG structure representing a tree likelihood (Fig. 2.3). Therefore, if the DAG is simplified as a simple binary tree, as in figure 3.2, then modifications of a branch length parameter can be identified as impacting the head vertex of its representative directed edge.

Based on this simplification, the best-case scenario is illustrated in figure 3.2a) and represents a sequence of perturbations ordered as the post-order depth-first traversal of the tree. This schedule has an average computational cost for each parameter defined as

$$C_b = \beta + \bar{h} \tag{3.12}$$

where $\bar{h}$ is the average tree height that was used to approximate the average PLU computational cost (Eq. (2.9)) and β is the average amount of edges separating two contiguous parameters in the sequence. Given that a perfect binary tree composed of vertices $\mathcal{V}$ has $|\mathcal{V}| - 1$ edges that are each traversed twice in a depth-first traversal, the average number of edges β separating two unvisited vertices can be defined as the amount of traversed edges divided by the number of vertices, as

$$\beta = 2(|\mathcal{V}| - 1)/|\mathcal{V}|. \tag{3.13}$$

The worst-case scenario, shown in figure 3.2b), identifies a traversal of the tree where the next vertex chosen is the one at the symmetrical opposite of the tree with respect to the root vertex. Whenever this opposite vertex has been already visited, its nearest unvisited neighbour is selected. This schedule has an average computational cost for

each parameter defined as

$$\mathcal{C}_w = 2\bar{h}$$

given that the perturbation of the previous step that must be cancelled and the one of the current step that must be applied are symmetrical.

Now that the first question is answered by defining both extreme schedules for this model, the second question is straightforward to answer. Indeed, the approximation of the maximal gain that can be expected from the perturbations scheduling is thus

$$\frac{\mathcal{C}_w}{\mathcal{C}_b} = \frac{2\bar{h}}{\beta + \bar{h}}.$$

When considering big trees, this expression can be further simplified as

$$\bar{h} \gg 1 \rightarrow |\mathcal{V}| \gg 1 \rightarrow \beta \approx 2 \rightarrow \frac{2\bar{h}}{\beta + \bar{h}} \approx 2.$$

For fully unbalanced binary tree, similar asymptotic behaviour should be observable. The best schedule has the same cost given that the cost of a depth-first traversal is identical. The worst schedule would have a similar cost expressed by a traversal moving from the further unvisited node, with regard to the root node, to the closest unvisited one and so on. Therefore, the same kind of maximal gain should be observed.

This expected maximal gain can only occur if the answer to the third question, regarding the choice of the reference schedule, is to chose the worst-case scenario previously defined. However, this choice is not representative of reality. Indeed, in reality, whenever the schedule would not be obtainable by any means, a sequence of perturbations would be blindly picked. The reference schedules used in the experiments were thus chosen by drawing permutations at random.

Now that the three points have been addressed, the experimental setting can be adequately described. The gradient computation was compared over the four following scheduling schemes, in order of expectation, from worst to best.

- **Random** sequences were used as reference scheduling approach. Three different random sequences were generated for each of the three replicates of each dataset.
- **Model-specific** was the scheduling based on the best-case scenario previously described. The five parameters of the evolutionary model were designated as the first of the sequence and were then followed by the branch length ordered as in a post-order depth-first traversal of the phylogenetic tree.
- A **unit** computational cost was assumed for each task. Using this information, the best solution found, using `LKH`, for the travelling salesman problem expressed by equation 3.11 was used as schedule.
- The average **time** measured for the execution of each task was used to define the schedule as in the previous scheme.

For each of these scheduling schemes, the measures used to define the speedup were the average measured times for twenty gradient approximations according to the computation sequence $\mathcal{S}$ (Eq. (3.5). Parallel likelihood evaluations were disabled in order to prevent the dynamic scheduling overhead to taint the measures. However, subtree compression was enabled given that it was proven to consistently improve the performance of `HOGAN`.

Scheduling using observed computational time

The first set of experiments aimed to assess the potential gain in performance coming from the best scheduling scheme, **time**, when compared to the reference one, **random**. In a first step, the speedup of the **time** scheme over the reference one was measured. As shown in figure 3.3a), the observed speedups increased with the tree size while they remained mostly constant with the codon numbers. This increase in speedup as a function of the tree size is explained by the augmenting difficulty of randomly obtaining a good perturbation permutation as the number of parameters increases.

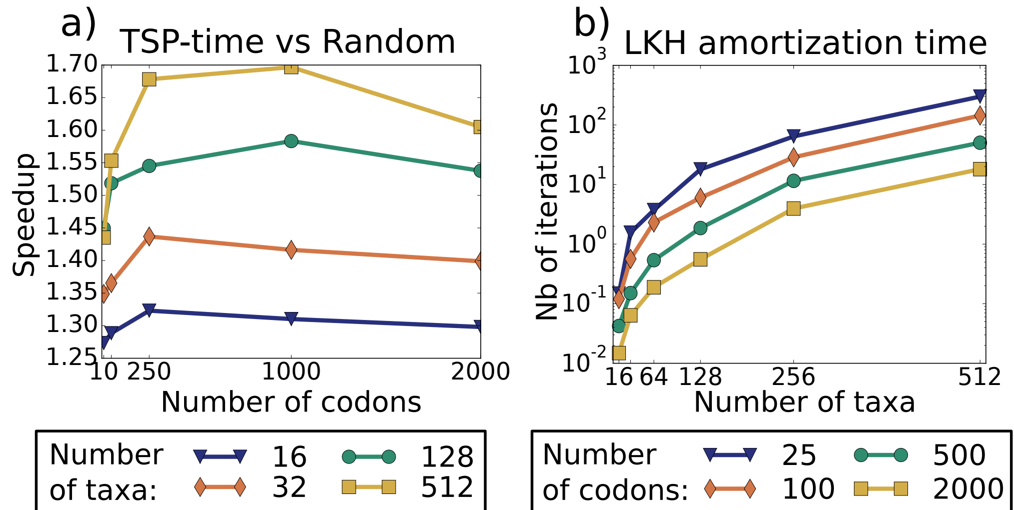


Figure 3.3.: Performance of the **time** scheduling scheme when compared to the reference one, **random**. The left figure shows the speedup when using the `LKH` to establish a schedule based on the observed computational time of tasks. The right figure defines the number of optimization steps required to negate the execution cost of `LKH`.

This trend also confirms the previously identified asymptotic upper bound on the gain defining a maximal twofold gain for big trees. However, this bound was obtained on the basis of various simplifications and assumptions such as the deliberate omission of the Matrix task and the choice to discard the effect of evolutionary model parameters. These rough approximations explain the smaller gains observed for sequences having few codons. Indeed, the likelihood evaluations, and thus the gradient approximations are

in this case dominated by the computational cost of the omitted Matrix and simplified Branch-Matrix tasks.

These significant speedups on the gradient approximations does not however tell the whole story. Indeed, the execution time of the `LKH` heuristic method used to solve the TSP instance representing the minimization problem is not taken into account. For that reason, the number of gradient approximations, or iterations, required to amortize this additional cost was estimated as

$$\frac{\mathcal{T}(LKH)}{\mathcal{T}(\mathbf{random}) - \mathcal{T}(\mathbf{time})}$$

where $\mathcal{T}$ represents the measured time for LKH and one gradient approximation with the **random** and **time** scheduling scheme.

This amortization time is illustrated in figure 3.3b) where it is shown to increase with the tree size and thus inherently with the size of the TSP instance. While this trend is problematic for the data of interest, the large ones, an opposite trend was observed in regard to the codon sequences size. Indeed, the computational cost of the gradient approximations increased with the size of the codon sequences and so did the gain $\mathcal{T}(random) - \mathcal{T}(time)$ while the `LKH` execution time $\mathcal{T}(LKH)$ remained constant.

In conclusion, on the data for which `HOGAN` was designed, the computation of gradient approximations were accelerated by upto a factor 1.7 when using the **time** scheduling scheme. The cost of defining a good schedule was shown to be amortized, in the worst cases, after a few tens of gradient approximations for problems having large codon sequences.

Performance of the unit and model-specific scheduling schemes

Now that the importance of having a good perturbation permutation has been demonstrated, the performance of the two other scheduling schemes is assessed by comparing them to the **time** scheduling scheme. The first of the two schemes **unit**, has the advantage of not depending on any measures, such as the observed computational time of tasks, and thus can be directly determined before any likelihood evaluations. The second one, the **model-specific** one, is even more interesting given that it can be directly derived from the structure of the phylogenetic tree and therefore can be obtained at the sole cost of a post-order depth-first tree traversal.

As illustrated in figure 3.4a), the **unit** scheme revealed itself however to be generally worse than the **time** one. While the tasks composing the likelihood of the branch-site model have initially an uniform computational costs, the use of the subtree compression changes the amount of CPVs to be computed in function of the data instance. Therefore, the **unit** scheduling scheme can lead to suboptimal perturbation permutations for this model and more importantly, could reveal totally inadequate for models having highly heterogeneous task computational costs.

The **model-specific** scheduling scheme also showed some performance variations when compared to the **time** one. However, in this case, these variations fitted, apart for the 16 taxa tree, into a $\pm 2\%$ range of the best permutation found. Therefore, considering

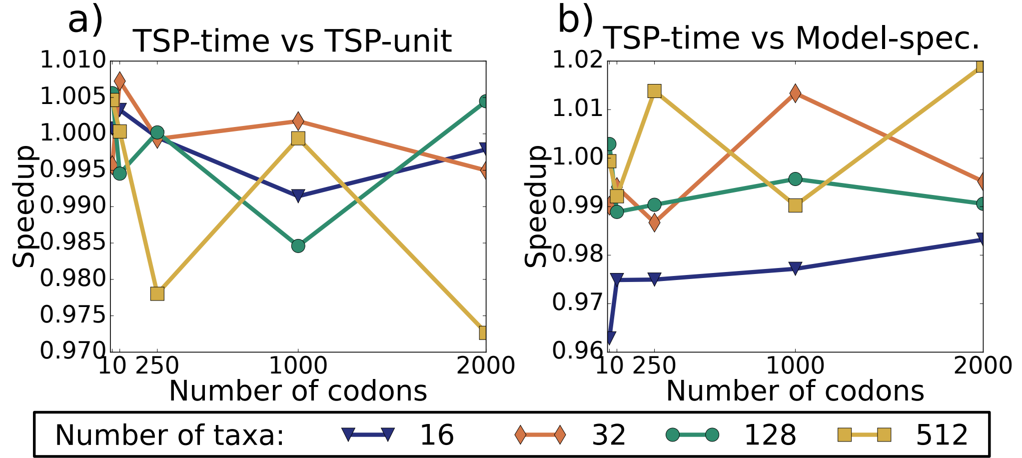


Figure 3.4.: Right and left figures illustrate the speedup of the **unit**, respectively **model-specific**, scheduling schemes when compared to the **time** one.

that this scheduling scheme has an extremely low cost when compared to a run of the Lin-Kernighan heuristic, it proves to be a good deterministic alternative to obtain good schedules for the branch-site model when compared to the **time** one.

3.4. Parallel gradient approximations

As defined previously, approximating the gradient requires the sequence of likelihood evaluations $\mathcal{S}$ (Eq. (3.5)) to be computed. Although these likelihood evaluations were demonstrated to express a dependency with respect to their computational cost, there exists no data dependency between each of them. Therefore, these likelihood evaluations can be computed independently and thus be distributed among several processors.

Ideally, each likelihood evaluations in the sequence $\mathcal{S}$ could be computed by a different processor and thus the gradient approximation would be accelerated by a factor $|\mathcal{S}|$. However, this is not the case in reality. The first obvious hurdle is that applying such approach to parameter-rich models would require supercomputers having thousands of processors or more. Assuming that such a supercomputer was available and used for that matter, the speedup would most likely not reach an $|\mathcal{S}|$-fold factor.

Indeed, the speedup,

$$Sp = \frac{\mathcal{T}_s}{\mathcal{T}_p}, \tag{3.14}$$

for this approach can be defined by simplifying the different kind of likelihood evaluations in the two already discussed categories: FLEs and PLUs. Based on this simplification, the computational time, or cost, of a sequential execution of sequence $\mathcal{S}$ can be defined as

$$\mathcal{T}_s = N_{\text{FLE}} \times \mathcal{T}_{\text{FLE}} + N_{\text{PLU}} \times \mathcal{T}_{\text{PLU}},$$
$$\text{with } N_{\text{FLE}} \geq 1 \text{ and } |\mathcal{S}| = N_{\text{FLE}} + N_{\text{PLU}}.$$

The time $\mathcal{T}$ for likelihood evaluations of categories FLE and PLU are defined by $\mathcal{T}_{FLE}$ and $\mathcal{T}_{PLU}$ respectively, while N_{FLE} and N_{PLU} denotes the number of evaluations of category FLE and PLU, respectively. The number of FLEs is at least one given that the first element of the sequence $\mathcal{S}$ is always a FLE.

Assuming that P processors are available then the time for a parallel computation of $\mathcal{S}$ can be given by

$$\mathcal{T}_p = \max\left[\mathcal{T}_{\text{FLE}}, \frac{N_{\text{FLE}}}{P} \times \mathcal{T}_{\text{FLE}} + \frac{N_{\text{PLU}}}{P} \times \mathcal{T}_{\text{PLU}}\right], \quad \text{with } P \leq |\mathcal{S}|.$$

This time is bounded by the time of a FLE, $\mathcal{T}_{\text{FLE}}$, given that a processor evaluating a single likelihood cannot reuse the partial results of the previous likelihood evaluation required for a PLU.

Whenever as many processors as needed are available, the speedup becomes

$$P = |\mathcal{S}| \rightarrow Sp = \frac{\mathcal{T}_s}{\mathcal{T}_p} \approx \frac{N_{\text{FLE}} \times \mathcal{T}_{\text{FLE}} + N_{\text{PLU}} \times \mathcal{T}_{\text{PLU}}}{\mathcal{T}_{\text{FLE}}} \tag{3.15}$$

$$\approx N_{\text{FLE}} + (|\mathcal{S}| - N_{\text{FLE}}) \times \frac{\mathcal{T}_{\text{PLU}}}{\mathcal{T}_{\text{FLE}}}, \tag{3.16}$$

Therefore, the ideal speedup can only be attained under the conditions that there is only FLEs or that the computational time of PLUs is identical to the one of FLEs. Given that both cases are quite improbable, this approach shows a first limitation.

In the more realistic case where the number of processors is limited by $P \ll |\mathcal{S}|$, then a load balancing approach could be adopted to level the cost of the tasks assigned to each processor. This NP-Complete problem, well known as the *independent tasks scheduling* problem [107], has been studied for decades and several heuristic approaches with an affordable complexity of $\mathcal{O}(n \log(n))$ have been designed to produce good task schedules. However, while the likelihood evaluations in $\mathcal{S}$ are independent data-wise, the problem at hand is far more complex given that their computational cost depends on the perturbations ordering $\mathcal{R}$ identified previously (Eq. (3.11)), making such heuristics not applicable.

A parallel can be drawn with the partitioning of an euclidean TSP instance, illustrated in figure 3.5. Partitioning the nodes, or cities, of a TSP instance such that the subtours length is balanced and minimal is a challenging task. The optimal tour $\mathcal{R}$ in figure a) does not provide an optimal ordering for the subtours. Indeed, the subtour of figure c) ordered according to $\mathcal{R}$ is longer that the one of figure d). Given that the *triangle inequality* holds in euclidean geometry, subtour c) can be proven to be at worst as long as the optimal tour $\mathcal{R}$. However, the best distance of a subtour is not known until the permutation of its nodes has been re-optimised.

Therefore, the difficulty of scheduling the sequence of likelihood evaluations $\mathcal{S}$ on P processors comes from the partitioning $\mathcal{D}$ of the computation composing $\mathcal{S}$ in multiple subsets that each must be optimally permuted such that the parallel computational cost

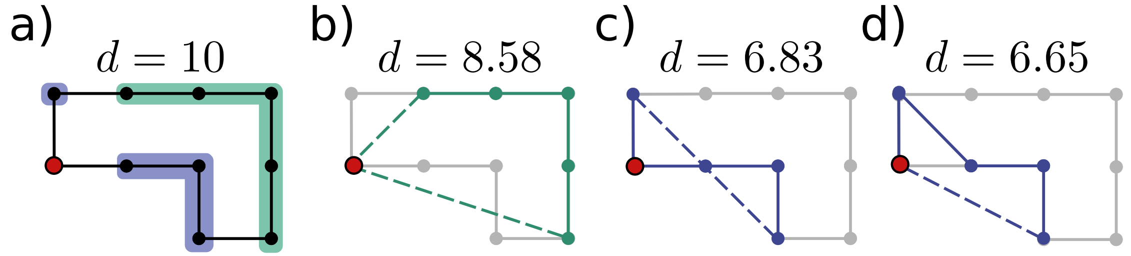


Figure 3.5.: Illustration of the partitioning of an euclidean TSP instance with unit distance and the starting node being the red one. Figure a) shows an optimal tour for a TSP with unit edge length. Nodes are divided in two partitions: blue and green. Keeping the ordering of the tour gives the two subtours of figures b) and c). However, as shown in figure d), these subtours are not obligatory optimal.

is minimised. The partitioning $\mathcal{D}$ in P subsets combined with the subset permutations $\mathcal{R}^{(p)}$ is defined as

$$\begin{aligned} \mathcal{D} = \Big\{ \mathcal{R}^{(p)} : p \in (1, .., P) \Big\} \\ \text{subject to} \quad & \emptyset \notin \mathcal{D}, \\ & \bigcup_{p \in (1,..,P)} \mathcal{R}^{(p)} = \mathcal{R}, \\ & \bigcap_{p \in (1,..,P)} \mathcal{R}^{(p)} = \emptyset. \end{aligned}$$

The computational cost of the perturbations $\mathcal{R}^{(p)}$ assigned to processor p can be defined by reusing the formalization of the computational cost $\mathcal{C}$ for a given perturbations permutation defined in equation (3.10) and by substituting $\mathcal{R}$ by $\mathcal{R}^{(p)}$. This formalization should be slightly adapted to reflect the fact that the first element of $\mathcal{S}$, representing the initial FLE, must only be computed once. All processors but one would then start at step $i = 1$; step that would, however, cost the same price as a FLE given that all the partial result of the directed task graph must be reinitialised. Since this change mainly acts as a small optimization[6], the initial definition of equation (3.10) remains however exact and is thus used in the forthcoming definitions to simplify the notation.

Using this cost $\mathcal{C}$, the total cost of the parallel computations in function of a task partitioning $\mathcal{D}$ and permutations $\mathcal{R}^{(p)}$ can be defined as

$$\sum_{p=1}^{P} \mathcal{C}(\mathcal{R}^{(p)}).$$

Minimising this cost is the aim of a generalization of the TSP, the multiple Travelling Salesman Problem [15] (mTSP) which is himself a special case of the more complex Vehicle Routing Problem [72] (VRP).

[6]This optimization is implemented in `HOGAN`.

While minimising this overall computational cost is rather important, the aim of assigning the likelihood evaluations of $\mathcal{S}$ to different processors is to reduce the effective time required to evaluate all likelihood functions in $\mathcal{S}$. This time, for a computation involving P processors, is identified by the time taken by the processor having the most work and thus being the last to finish its computation. Therefore, the optimal tasks partitioning $\mathcal{D}$ and permutations $\mathcal{R}^{(p)}$ for P processors are defined by

$$\underset{P,\mathcal{R}^{(\cdot)},\mathcal{D}}{\arg\min}\left[\beta(P)+\max_{p}\left(\mathcal{C}(\mathcal{R}^{(p)})\right)\right]. \tag{3.17}$$

The previous definition also optimises the number of processors P by adding a term $\beta(P)$ identifying the communication overhead in function of P to the maximal cost $\mathcal{C}$. However, given that this problem complexity is already enough without the optimization of P, the communication overhead is assumed to be insignificant with respect to the computational cost of tasks and thus P is defined by the amount of computing resources supplied.

Given that the task scheduling problem and the TSP are NP-Hard, this problem combining both of them shares their complexity. The total number of possible solutions gives a good overview of the task at hand. Therefore, this number is defined by using the number of permutations of a set defined in equation (3.9) and then by using the *Stirling number of second kind*,

$$\left\{{n \atop k}\right\}=\frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^{n},$$

that gives the number of possible ways to partition a set of n object into k nonempty subsets.

The number of possible partitioning combined with the respective amount of permutations for each subset then defines the total number of possible solutions as

$$\sum_{\mathcal{D}\in\mathbb{D}}\left[\prod_{\mathcal{R}^{(p)}\in\mathcal{D}}\binom{|\mathcal{R}^{(p)}|}{|\mathcal{R}^{(p)}|}\right]=\sum_{\mathcal{D}\in\mathbb{D}}\left[\prod_{\mathcal{R}^{(p)}\in\mathcal{D}}\left(|\mathcal{R}^{(p)}|\right)!\right],$$
$$\text{with } |\mathbb{D}|=\left\{{|\mathcal{R}| \atop P}\right\}.$$

Considering that each permutation $\mathcal{R}^{(p)}$ can be defined independently from the others for a given assignment $\mathcal{D}$, the complexity of a brute force approach can be defined as

$$\sum_{\mathcal{D}\in\mathbb{D}}\left[\sum_{\mathcal{R}^{(p)}\in\mathcal{D}}\left(|\mathcal{R}^{(p)}|\right)!\right].$$

Computing these numbers is already a challenge in itself given that the enumeration of the possible partitioning must be computed to evaluate the second term. Therefore, this hard scheduling problem is approached in `HOGAN` using a custom heuristic method defined in the next section.

3.4.1. Scheduling and load balancing

Designing a good heuristic method for the problem previously defined is difficult given that the cost of the PLU representing the perturbation computation depends on the partition $\mathcal{R}^{(i)}$ on which it is affected and also on its position in the said partition. Moreover, in function of the position chosen it not only modifies the PLU cost, but also the one of the subsequent PLU in the partition. Therefore, even if a correct initial partitioning can be defined, then swapping some perturbations such as to balance the load may have undesirable effects.

To be sure of the effects of a PLU swap, both the source and destination partitions must have their task sequence optimised to find again a good permutation. However, even if it was previously defined that good heuristics exist, performing several times such operation would quickly lead to an additional computational cost that would require a significant amount of optimization steps to be negated.

Algorithm 7 Load balancing strategy for parallel gradient approximations.
The costs are evaluated using the different *levels* (overall partitioning, partitions, PLUs) expressed in equation (3.17).

`// init by partitioning a good sequential scheduling`
$\mathcal{D} = \text{init}(\mathcal{R})$
`// Repeat while the partitioning is improving`
repeat
  $\hat{\mathcal{D}} = \mathcal{D}$
  `// define slowest and fastest tasks partitions`
  $\mathcal{R}^{(s)} = \arg\max_{\mathcal{R}^{(i)} \in \mathcal{D}} \left[\text{cost}\left(\mathcal{R}^{(i)}\right)\right]$
  $\mathcal{R}^{(f)} = \arg\min_{\mathcal{R}^{(i)} \in \mathcal{D}} \left[\text{cost}\left(\mathcal{R}^{(i)}\right)\right]$
  `// define cost imbalance between both partitions`
  $\Delta\mathcal{R} = \text{cost}\left(\mathcal{R}^{(s)}\right) - \text{cost}\left(\mathcal{R}^{(f)}\right)$
  `// find task potentially levelling both partitions`
  $t = \arg\min_{j \in \mathcal{R}^{(s)}} |\Delta\mathcal{R}/2 - \text{cost}(j)|$
  `// if` t `can improve the assignment, update it`
  if $\text{cost}(t) < \Delta\mathcal{R}$ **then**
    $\mathcal{R}^{(s)} = \mathcal{R}^{(s)} \setminus t$
    $\mathcal{R}^{(f)} = \mathcal{R}^{(f)} \cup t$
    `// Place` t `in` $\mathcal{R}^{(f)}$ `according to the order expressed by` $\mathcal{R}$
    $\text{order}(\mathcal{R}^{(f)}|\mathcal{R})$
  end if
until $\text{cost}(\mathcal{D}) \geq \text{cost}(\hat{\mathcal{D}})$
return $\hat{\mathcal{D}}$

In order to address this issue, a rather simple heuristic method has been designed for `HOGAN`. This method is based on two elements: a good sequential perturbation schedule,

plus a basic load balancing strategy. Indeed, using these two strategies, the heuristic method defines a correct schedule in three steps.

1. A good sequential perturbations schedule $\mathcal{R}$ is defined using the approaches defined in the previous section. If a **model-specific** scheduling scheme exists for the considered model, then it is employed. Otherwise, the **time** scheduling scheme previously defined is used.

2. Using this permutation $\mathcal{R}$, the initial partitioning $\mathcal{D}$ is determined. For that matter, equally sized and contiguous subsets of $\mathcal{R}$ are attributed to the P partitions. Given that the *triangle inequality* holds for the problem at hand (see below and Thm. 5), this step ensures that an adequate initial partitioning with decent permutations is obtained as a starting point.

3. This initial partitioning $\mathcal{D}$ is refined iteratively using the strategy described in algorithm 7. This strategy tries to improve the partitioning by levelling the cost of the most unbalanced partitions. For that matter, the task being the best candidate for the levelling is swapped from the *slowest* to the *fastest* partition. In order to avoid the optimization of these permutations of partitions, partitions in $\mathcal{D}$ are always sorted accordingly to the order expressed in $\mathcal{R}$.

This heuristic gives suboptimal solutions for the problem defined in equation (3.17) because the permutation of each partition is not individually optimised. However, theorem 5 (Apx. B.1) proves that the *strict triangle inequality* holds for the metric used to define this permutation problem (Eqs. (3.10) and 3.11). This important property of the metric has played a key role in defining several theoretic results for the TSP. For instance, solutions within $1 + \epsilon$ of the optimal have been shown to be theoretically obtainable in polynomial time for instances of the TSP guaranteeing the triangle inequality [8].

In the context of the heuristic presented here, this property guarantees that partitions $\mathcal{R}^{(p)}$ sorted according to the order expressed in a good sequential permutation $\mathcal{R}$ have a cost $\mathcal{C}$ (Eq. (3.10)) strictly inferior to the one of the full permutation, such that

$$\mathcal{R}^{(p)} \subsetneq \mathcal{R} \to \mathcal{C}\left(\mathcal{R}^{(p)}\right) < \mathcal{C}\left(\mathcal{R}\right).$$

Therefore, the obtained partitioning should always lead to, at least, a small performance gain. While this heuristic produced good schedules for the branch-site model, as demonstrated in the following section, a more carefully designed heuristic method could be a good addition to `HOGAN` in order to further improve the quality of the partitioning $\mathcal{D}$ and the permutation of its inherent partitions.

3.4.2. Experiments on the branch-site model (cont.)

The potential gains coming from the parallel gradient approximations, and thus, of the previously described heuristic method are assessed in this section. This analysis wraps up the experiments on the branch-site model defined in equation (1.8). Therefore, the dataset remained the same as in the previous experiments, as well as the measurement

protocol that was defined during the analysis of the scheduling of gradient approximations.

In the first phase of the analysis, the heuristic scheduling method was analysed by comparing it with three other simpler approaches. These experiments, using only few computational resources (upto 8 processors), aimed at separating the effects of the two main components: the sequential perturbation schedule and the load balancing.

- The first method represented the case where none of the components were used. Starting from a random permutation of perturbations, this method partitioned equally these perturbations in P subsets.
- The second method improved the first one by applying the load balancing defined in algorithm 7 after the partitioning.
- The third method partitioned a good sequential perturbations schedule defined using the **time** scheduling scheme presented in the previous section but did not apply the load balancing algorithm.

In the second analysis phase, the experiments were conducted on a larger scale with respect to the amount of computational resources used. While parallel likelihood were benefiting from a shared memory implementation, the parallel computation of gradient approximations, requiring fewer communications, is implemented using MPI in order to profit from the inherent scaling of distributed memory architectures. Therefore, this second set of experiments assessed the scaling of `HOGAN` with as much as 128 processors on the branch-site model by measuring the performance of various combinations of parallel likelihood and parallel gradient approximations.

Performance model

The performance gains expected for both sets of experiments can be defined using the speedup equation (3.14). Assuming a phylogenetic tree structure close to a perfect binary tree having S taxa, the branch-site model counts 5 parameters that require FLEs, and $2S-2$ branch length parameters that require PLUs. Furthermore, during the previous analysis of the tree likelihood, the expected computational cost for such trees were approximated as $2S-1$ for FLEs and as $\bar{h}+\beta$ (Eq. (3.12)), the average tree height plus a small constant representing the difference between two parameter perturbations, for PLUs.

Therefore by using

$$\begin{aligned} N_{\text{FLE}} &= 5, \\ N_{\text{PLU}} &= 2S-2, \\ \mathcal{T}_{\text{FLE}} &\approx 2S-1, \\ \mathcal{T}_{\text{PLU}} &\approx \beta + \log_2(S) - 1, \end{aligned}$$

the expected speedup of parallel gradient approximation can be estimated. These values for a number of processors are given in the first and second rows of the table 3.1 for the

Table 3.1.: Expected speedups for joint parallel gradient approximations and parallel likelihood/gradient computations for a tree having 512 taxa.

P_{lik}	β	Total nb. of processors ($P_{lik} \times P_{grad}$)								
		1	4	8	16	24	32	64	96	128
1	2	1.0	4.0	6.6	9.2	10.6	11.4	13.0	13.6	13.9
	5	1.0	4.0	6.8	9.9	11.7	12.8	15.0	15.8	16.3
4	2	-	2.7	5.4	11.0	15.6	18.7	26.9	31.6	34.5
	5	-	2.7	5.4	10.8	15.5	19.0	28.6	34.3	38.2

optimal schedule $\beta = 2$ (Eq. (3.13)) and, an average scheduling represented by $\beta = 5$ chosen to be roughly equivalent to setting a cost of $\mathcal{T}_{\text{PLU}} \approx 1.5\bar{h}$ when $S = 512$.

When P_{grad} processors are attributed for parallel gradient approximations and P_{lik} for parallel likelihood evaluations, the computational costs of PLUs and FLEs have to be adapted to represent the speedup $Sp_{(\cdot)}$ gained. Therefore, the time for a parallel execution $\mathcal{T}_p$ is redefined as

$$\begin{aligned}
\hat{\mathcal{T}}_p = \max &\left[\hat{\mathcal{T}}_{\text{FLE}}, \frac{N_{\text{FLE}}}{P_{grad}} \times \hat{\mathcal{T}}_{\text{FLE}} + \frac{N_{\text{PLU}}}{P_{grad}} \times \hat{\mathcal{T}}_{\text{PLU}}\right], \text{ with} \\
& P_{grad} \leq |\mathcal{S}|, \\
& \hat{\mathcal{T}}_{\text{FLE}} = \frac{\mathcal{T}_{\text{FLE}}}{Sp_{\text{FLE}}\left(P_{lik}\right)}, \\
& \hat{\mathcal{T}}_{\text{PLU}} = \frac{\mathcal{T}_{\text{PLU}}}{Sp_{\text{PLU}}\left(P_{lik}\right)}.
\end{aligned}$$

The values for Sp_{FLE} and Sp_{PLU} are directly derived from the observed speedups during the experiments on parallel likelihood evaluations (Fig. 2.14). Using this new definition of the time of a parallel execution, the expected speedup $\mathcal{T}_S/\hat{\mathcal{T}}_P$ for $P_{lik} = 4$ is reported in table 3.1 in the third and forth rows with the two different values of β.

The expected speedup depends on β and on P_{lik} as shown in table 3.1. The previously identified asymptotic speedup (Eq. (3.15)) readily explains these dependencies. Indeed, for the branch-site model, N_{FLE} and N_{PLU} are constant, the term of importance is thus the ratio between the times, or costs, $\mathcal{T}_{\text{PLU}}$ and $\mathcal{T}_{\text{FLE}}$. This ratio defines the ideal speedup obtained when both costs are equal. Therefore, augmenting β and thus $\mathcal{T}_{\text{PLU}}$, pushes the expected speedup toward the ideal one making thus the parallel approximation of gradients resilient to a decrease in the scheduling quality and thus an increase in $\mathcal{T}_{\text{PLU}}$.

The effect of the combined use of parallel likelihood and gradient approximations is even more interesting. Indeed, the scaling of FLEs was previously shown to be better than the one of PLUs when using $P_{lik} > 1$ (Fig. 2.14). Therefore, using multiple processors to compute the likelihoods has the effect to level the costs $\mathcal{T}_{\text{PLU}}$ and $\mathcal{T}_{\text{FLE}}$ by further speeding up the FLEs. In other words, this joint parallel approach has the potential to be more efficient than the separate application of its composing parallel methods.

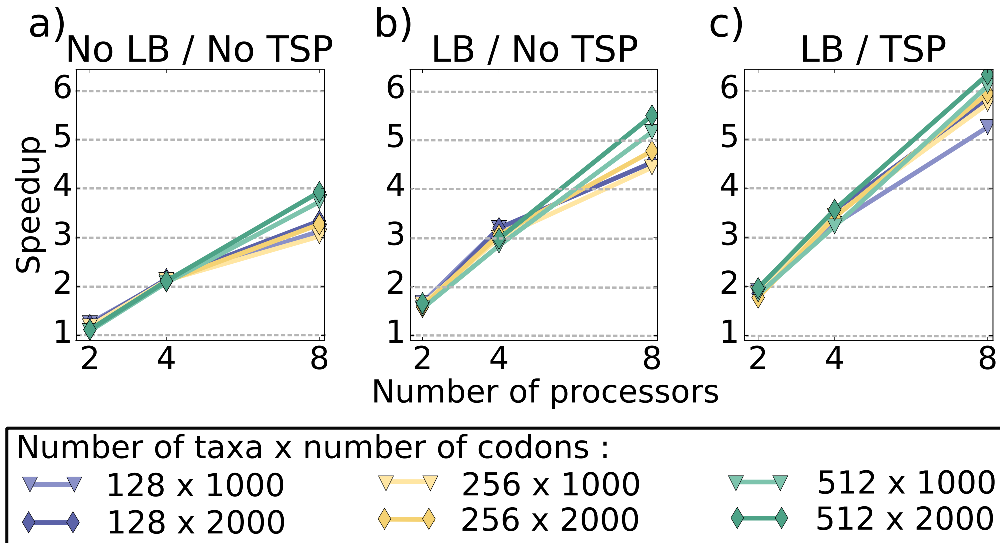


Figure 3.6.: Comparison of three different scheduling schemes for the parallel approximations of gradient. These scheduling scheme are characterised by the use, or not, of load balancing (LB) and a good initial sequential schedule (TSP). The sequential reference time used for the speedup was obtained with the model-specific approach.

These theoretic properties are challenged against the experimental results presented in the forthcoming sections.

Assessing the performance of the heuristic method

The heuristic used for the scheduling of the parallel gradient approximations implemented in `HOGAN` is analysed in this section by monitoring the effects of its two main components. The first experiment illustrated the most basic approach, where neither the load balancing nor a good initial sequential schedule was used. While this approach scaled decently with the number of processors, its speedup, shown in figure 3.6a), was half the expected speedup (Table 3.1). These results were clearly expected given that neither the partitioning nor the partitions permutations were wisely chosen. The decent speedup scaling in these circumstances was more surprising. However, this trend confirms the resiliency to bad schedules previously identified on the performance model.

Upon the addition of load balancing, the speedup of this simplistic approach was significantly enhanced as illustrated in figure 3.6b). Indeed, the improved partitioning defined by load balancing coupled with the resiliency to average perturbation permutations was directly reflected in significant performance gains. When compared to the expected speedup, these gains nearly reached an optimal value when two processors were used. However, for higher number of processors the performance was still below the prediction of the performance model.

The heuristic implemented in `HOGAN` further improved the observed speedup by using

a good sequential schedule as starting point. The initial partitioning derived from this good schedule seemed to be already well balanced. Indeed, the contribution of the load balancing only led to an average enhancement of the speedup of $4 \pm 3\%$. While the performance gains shown in figure 3.6c) are once again closer to the expected ones, there remains a gap between both of them that is readily explainable by the multiple approximations made for the performance model as well as by the suboptimal schedule defined by the heuristic method.

Finally, while these measures are not sufficient to give an adequate assessment on the scaling of the heuristic method, they showed the parallel approximation of gradient to be rather robust to the size of the data. Indeed, when compared to the variation of the speedup monitored during the experiments on the parallel likelihood, these results appeared to be nearly independent on the size of the problem as well as the use of subtree compression.

Combined parallel likelihood and gradient

In these final experiments on the branch-site model the combination of the parallel likelihood evaluations on shared memory and the parallel gradient approximations on distributed memory is evaluated. This mixed parallelism was analysed under different couplings defined first by the number of processors dedicated to the parallel likelihood evaluations $P_{lik} \in (1, 2, 4, 8)$ and then by the number of processors dedicated to parallel gradient $P_{grad} \in (1, 4, 8, 16, 24, 32, 64, 96, 128)$ with, however, a limit of 128 to the total number of processors.

Only one data was selected for this benchmark in order to reduce its computational cost. The three replicates of the biggest data, the one having 512 taxa and 2000 codons, were thus used to measure the speedup of this mixed parallelism approach. These performance gains, as shown in figure 3.7a), scaled with the amount of processors and were close to the expected speedups represented on the figure by blurred blue and green line for 1 processor and respectively 4 processors.

These observed speedups were far from linear as the performance model predicted. However this model also defined that this coupling of parallel approach could prove more efficient than their independent application. Therefore, the efficiency of this coupling, defined as

$$\frac{Sp}{P_{lik} \times P_{grad}},$$

is illustrated in figure 3.7b). The coupling of $P_{lik} = 4$ processors for the likelihood evaluations with $P_{grad} \geq 4$ processors for the gradient approximations was revealed to be the most efficient approach and thus confirmed the theoretic predictions. Indeed, the measured efficiency of parallel likelihood evaluations quickly crashed with the increase of processors while the one of the parallel gradient approximations remained the best until 16 processors were used and, from this point, started to fall faster than the previously identified coupling.

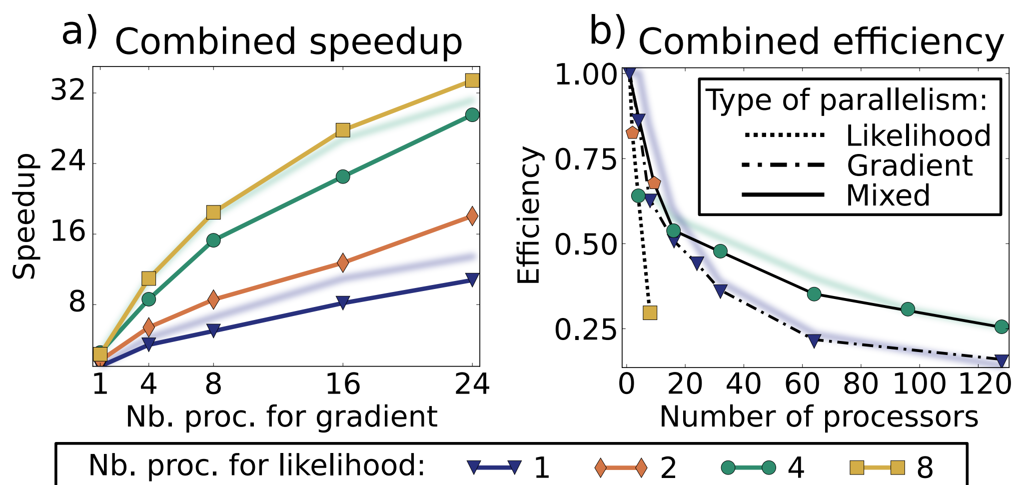


Figure 3.7.: Performance of the mixed likelihood/gradient parallelism. Left figure shows the speedup in function of P_{grad} when both methods are combined. Right figure illustrates the ratio speedup to processors, know as efficiency, of this mixed approach in function of the total number of processors used: $P_{lik} \times P_{grad}$. The blue and green blurred line represents the expected speedup and efficiency, respectively, based on speedup measures of table 3.1 with $\beta = 2$.

Concluding discussion on the branch-site model

In order to conclude these experiments on the branch-site model, the total improvements over the state-of-the-art implementation `FastCodeML` is summarized. In chapter 2, the implementation of PLUs in `HOGAN`, jointly with some other improvements, were shown to accelerate the evaluations of likelihoods by upto three orders of magnitude when compared to `FastCodeML`. This speedup of the likelihood evaluation was furthermore identified to grow with the size of the data. On top of this first performance gain, the use of parallel likelihood evaluations further sped up these computations by, on average, a threefold factor when 8 processors were used.

In the current chapter, a scheme defining a good schedule for the computation of the gradient approximation was presented. This scheme optimised the use of PLUs making thus the computations of the gradient approximation proportional to the one of the average likelihood evaluations presented in the previous chapter. Using this scheme, a heuristic method for the partitioning of gradient approximations for multiple processors was designed. This method, jointly with the application of parallel likelihood, accelerated the computations of the gradient approximations by a twentyfold factor with 36 processors (50% efficiency) and upto to a thirtyfold factor with 128 processors.

To make this comparison fair, the documented speedups [127] on the likelihood evaluations of `FastCodeML` are taken into account. With a maximal amount of 12 processors, `FastCodeML` was shown to accelerate the likelihood evaluations by a eightfold factor. Therefore the following projection of the performance gains can be presumed:

- In a sequential setting, `HOGAN` was shown to be three order of magnitude faster than `FastCodeML` on large data instance thanks to a reduction of the likelihood evaluation complexity in function of number of taxa S from an initial $\mathcal{O}(S)$ to $\mathcal{O}(log(S))$ in the best-case scenario.

- In a parallel setting and with an equal number of 12 processors bounded by the implementation in shared memory of `FastCodeML`, the precedent scale of gain were preserved.

- When using more than 12 processors, the improved scaling of the mixed parallelism approach implemented in `HOGAN` showed additional performance increase. These gains were shown to further double when using upto 36 processors and triple when using upto hundredth of processors.

In conclusion, `HOGAN` was shown to have the potential of speeding up statistical analysis of the branch-site model by hundredfold to thousandfold factors when compared to the fastest existing implementation.

3.5. Summary

In this chapter, means of improving the efficiency of ML methods, more precisely gradient-based continuous optimization methods such as BFGS or LBFGS, are studied. Based on the target applications for HOGAN, the choice of the method of finite differences for the gradient approximation is justified. This method highly benefits from PLUs given that the gradient is estimated by using univariate perturbations on the parameters.

The formalization, with respect to the PLUs, of this sequence of univariate perturbations leads to two model-generic optimization schemes, a sequential and a parallel one, that can be used to further improve the efficiency of gradient approximations. Both strategies are illustrated on the branch-site model detecting positive selection.

Model-generic

Two novel model-generic methods enhancing the efficiency of the gradient approximation are proposed. The first method results from the study of the sequential approximation of the gradient:

- Sequences of perturbations are formalised by using the PLUs definition.
- The scheduling of these perturbations is modelled as a permutation problem which is then identified as an instance of the TSP.
- Estimations of the task cost, as presented in chapter 2, enables the definition of a strategy providing good perturbation scheduling.
- This scheme is implemented with the help of an existing state-of-the-art heuristic for the TSP, LKH.

The second method results from a study on the parallel approximation of the gradient:

- An ideal performance model is derived for the parallel estimation of gradients.
- This performance model is proved unrealistic by formalising the joint problem of partitioning and scheduling the sequence of perturbations.
- A greedy heuristic is proposed for the optimization of this complex combinatorial problem.
- This heuristic is proved to always result in performance gains when using the sequential schedule obtained from the first method as starting point.

Model-specific

The potential performance gains of the sequential method are studied on the phylogeny based likelihoods and an asymptotic theoretic upper bound, with respect to S, is derived. Furthermore, a good model-specific perturbations scheduling is identified for the phylogeny based model.

The sequential and parallel methods are validated on the branch-site model detecting positive selection with a large variety of datasets. The results of the experiments with the sequential method are the following:

- This method accelerates upto 1.7 times the gradient approximations when compared to a random perturbation scheduling and its additional computational cost is shown to be quickly compensated.
- The model-specific scheduling obtains similar performance gains when compared to this method and, conversely to the model-generic approach, does not incur any additional computational cost.

The study of the parallel method on the branch-site model consists in the following results:

- The greedy heuristic is studied on a selection of datasets. This method is shown to significantly improve the scaling of the parallel gradient approximation.
- A joint performance model of the parallel gradient approximation and the parallel likelihood evaluation is derived.
- This performance model is used to study the combined scaling of these methods and these theoretic results are compared with empirical results:
  - The observed speedup are concordant with the performance model and enables to predict the most efficient assignment of computational resources.
  - On a large dataset, twentyfold and thirtyfold speedups are measured by combining the use of 4 processors for parallel likelihood evaluations with 9 and 32 parallel gradient estimators for a total of $4 \times 9 = 36$ and $4 \times 32 = 128$ processors, respectively.

This study finalises the experiments on the branch-site model by concluding that, by combining all the model-generic methods presented in the previous and current chapter, `HOGAN` has the potential to accelerate ML analysis on this model and synthetic datasets by upto three order of magnitude when compared to the state-of-the-art software `FastCodeML`.

4. Bayesian inference methods

Bayesian inference derives the posterior probability distribution of the parameters θ of a given statistical model after taking into account the observed data X. This distribution is defined as the joint distribution obtained by combining the prior probability of parameters θ and the likelihood, derived from the model, of observing data X given parameters θ. The *Bayes Theorem* determines the posterior probability as

$$p(\theta|X) = \frac{p(\theta)f(X|\theta)}{\int p(\theta)f(X|\theta)d\theta}, \tag{4.1}$$

with $p(\theta)$ as the prior probability and $f(X|\theta)$ as the likelihood function.

While being a powerful tool for inference, this approach is limited by the difficulty of estimating the posterior distribution. The MCMC method designed by Metropolis *et al.* [84] tackled this limitation by making possible the numerical approximation, or sampling, of high-dimensional integrals. Enriched through the years by further theoretical and computational developments, this method gained in popularity such that nowadays model inference and hypothesis testing is widely performed using Bayesian inference. Evolutionary biology particularly benefited from this development, however the design of more complex and realistic models and the ever growing availability of novel data, such as genomic data, is pushing the limits of the current use of the MCMC method. In addition to the expensive computational cost of models, the burden placed on the user to define adequate parameter moves to efficiently explore the parameter space results usually in sub-optimal sampling schemes.

Indeed, MCMC methods approximate the posterior distribution by sampling the state of a Markov chain constructed such as to have this exact distribution as equilibrium distribution. An important element in this construction is the proposal kernel that suggests new parameter values that are then judged as fitting, or not, the target distribution. Therefore, this proposal kernel determines, in function of its closeness or adequacy to the unknown posterior distribution, the efficiency of a MCMC sampler. Paradoxically, the complex choice of this proposal kernel depends on the user that must then decide which form of proposal distribution, as well as its scaling, is adequate for each of the statistical model parameters. In front of such a daunting task, users usually have no other choice than to use an arbitrarily scaled univariate proposal that updates parameters one at a time; choice that results in a suboptimal sampling efficiency.

This chapter begins by giving a short introduction of the MCMC methods and more particularly of one of its variant, the Metropolis-Hastings algorithm. Then, a parallel MCMC method built with a novel combination of enhancements designed for the sampling of parameter-rich and complex models is presented. The theoretical definition of this model-generic method is detailed in a first section that is followed by a second one

describing the required steps to obtain an efficient implementation of the proposed algorithm. The resulting implementation in HOGAN is then confronted to multiple biological applications, such as the reconstruction of phylogenetic tree on which it is shown to outperform a well-known state of the art software for Bayesian inference of phylogeny, MrBayes, by converging upto twenty times faster to the correct tree distribution. Finally, the DAG representation of statistical models used in HOGAN again proves its richness by enabling the automation of a complex task: the choice of parameter blocks to update.

4.1. A short introduction to MCMC methods

This section gives a brief overview, largely inspired from the book *Markov Chain Monte Carlo, Stochastic Simulation for Bayesian Inference* [39], of the estimation of posterior distribution using the MCMC method. The main concept is defined and then some important considerations regarding the use of this method in practice are discussed.

4.1.1. Definition

Given a statistical model, Bayesian inference consists in analysing the posterior distribution of the model parameters[1] $\theta \in \mathbb{R}^d$ defined in equation (4.1), or its unnormalized density, defined as

$$\pi(\theta) \propto p(\theta) f(X|\theta), \tag{4.2}$$

after having observed data X. In theory, the posterior distribution is always available, but in parameter-rich and complex models, the required analytic computations are often intractable.

MCMC methods are thus employed to approximate the posterior distribution by sampling the state of a Markov chain having $\pi(\theta)$ as stationary distribution. Constructing a Markov chain that achieves this property is done by considering a reversible Markov chain whose transition kernel $p(\theta, \phi)$ satisfies the detailed balance equation

$$\pi(\theta)p(\theta, \phi) = \pi(\phi)p(\phi, \theta) \qquad \forall\ (\theta, \phi)$$

The transition kernel $p(\theta, \phi)$ consists of two elements: an arbitrary transition kernel $q(\theta, \phi)$, also known as proposal kernel, and an acceptance probability $\alpha(\theta, \phi)$ such that the probability of leaving the state θ is given by

$$p(\theta, \phi) = q(\theta, \phi)\alpha(\theta, \phi) \qquad \text{with } \theta \neq \phi.$$

Therefore, the probability of staying in the same state is given by

$$p(\theta, \theta) = 1 - \int q(\theta, \phi)\alpha(\theta, \phi) d\phi$$

[1] For simplicity's sake, these parameters are considered as being continuous, however it also holds for discrete parameters.

Hastings [56] proposed to define the acceptance probability such that its combination with the proposal kernel ensures the detailed balance equation. The resulting acceptance probability is commonly defined as

$$\alpha(\theta, \phi) = \min\left(1, \frac{\pi(\phi)q(\phi, \theta)}{\pi(\theta)q(\theta, \phi)}\right). \tag{4.3}$$

Whenever the proposal kernel q is symmetrical, this probability simplifies to the original definition of Metropolis *et al.* [84],

$$\alpha(\theta, \phi) = \min\left(1, \frac{\pi(\phi)}{\pi(\theta)}\right) \qquad \text{with } q(\phi, \theta) = q(\theta, \phi).$$

4.1.2. Practical aspects

Samplers based on the M-H algorithm form an important subset of the MCMC method and can be readily implemented based on the previous definitions. Indeed, given initial parameter values $\theta^{(0)}$ at step $k = 0$ and a user defined proposal kernel q, a MCMC process drawing samples from the distribution π can be obtained by iterating through the following steps:

1. Increment the iteration counter, $k = k + 1$.
2. Propose new values $\phi^{(k)}$ generated using density $q\left(\theta^{(k-1)}, \cdot\right)$.
3. Evaluate the acceptance probability $\alpha\left(\theta^{(k-1)}, \phi^{(k)}\right)$ according to equation (4.3). This step requires the evaluation of the posterior probability of parameter $\phi^{(k)}$ according to equation (4.2).
4. Draw a number $u \in [0, 1]$ from a independent uniform distribution.
    - If $\alpha\left(\theta^{(k-1)}, \phi^{(k)}\right) \geq u$, accept the move: $\theta^{(k)} = \phi^{(k)}$.
    - Else, reject it: $\theta^{(k)} = \theta^{(k-1)}$.

This sequence of steps is repeated until the approximation of $\pi(\theta)$ is deemed accurate enough.

For parameter-rich and complex models, defining a single proposal kernel q that moves adequately all the parameters at once may prove to be a daunting task. Fortunately, the parameters θ can also be updated separately [125, 39]. Assuming a set of user defined univariate proposal kernels $\{q_i : i \in (1, .., d)\}$ that each acts on a parameter θ_i, the sampling scheme previously described can be adapted by slightly changing steps 2), 3) and 4).

The step 2) must first select a parameter θ_i. This parameter choice can be random or based on a fixed predefined order: e.g $1 \to 2 \to .. \to d$. The proposal kernel q_i is

then used to suggest a new value for parameter θ_i. The steps 3) and 4) apply the same operations using the parameter-wise acceptance probability defined as

$$\alpha_i\left(\theta_i^{(k-1)},\phi_i^{(k)}\right)=\min\left(1,\frac{\pi\left(\phi_i^{(k)}\right)q\left(\phi_i^{(k)},\theta_i^{(k-1)}\right)}{\pi\left(\theta_i^{(k-1)}\right)q\left(\theta_i^{(k-1)},\phi_i^{(k)}\right)}\right).$$

While this scheme simplifies the choice of the kernels q_i, at least d iterations are required to propose a move on each of the d parameters and thus as many likelihoods are evaluated. However, this simplification greatly eases the use of the M-H algorithm and is widely employed in computational evolutionary biology. For instance, `PyRate` and `MrBayes` use univariate proposal kernels.

The sampling of the posterior distribution π of an independent bivariate normal model, defined in section 1.3.1, with this algorithm is illustrated in figure 4.1. For such a simple model, several thousands of iterations, and thus likelihood evaluations, were required to obtain a good approximation of the distribution. Moreover, the proposal kernels were chosen to produce an efficient sampling scheme and the starting parameter values $\theta^{(0)}$ were directly drawn from the posterior distribution.

These last considerations lead directly to two important properties of a MCMC process that play a key role in the number of iterations, and thus time, required to get a correct approximation of a posterior distribution. The first property is the convergence time required for the Markov chain to *reach its equilibrium distribution*; time that is implicitly, but not exclusively, determined by the starting values $\theta^{(0)}$. The second property is the sampling effectiveness, or *mixing*, of the MCMC process which is determined by the mobility of the Markov chain in the parameter space and is thus highly dependent on the choice of the proposal kernels.

Convergence of the chain

Depending on the starting parameter values $\theta^{(0)}$ and as well on the proposal kernel chosen, the convergence of a chain toward the equilibrium distribution can take a considerable time as illustrated, in figure 4.2. This first phase, known as *burn-in* phase, is characterised by observed samples that are not representative of the High Probability Density (HPD) of the posterior distribution and that are, for that reason, not taken into account for the approximation of $\pi(\theta)$. Therefore, given that the burn-in phase represents a waste of computational resources and time, an important properties of a MCMC algorithm is to have a fast convergence.

In addition of representing a waste of resources, this phase highly limits the perspective of parallelism for MCMC methods. Indeed, the concept of MCMC algorithm previously defined is inherently sequential given that each step k depends on the previous state $\theta^{(k-1)}$ of the chain; state that is furthermore defined stochastically. However, in theory, a simple approach could enable the use of P processors to estimate the posterior distribution. Assuming that several independent chains were run on these P processors, the samples resulting from each of these chains could be readily concatenated, given that each sample is considered as independently drawn from the posterior distribution.

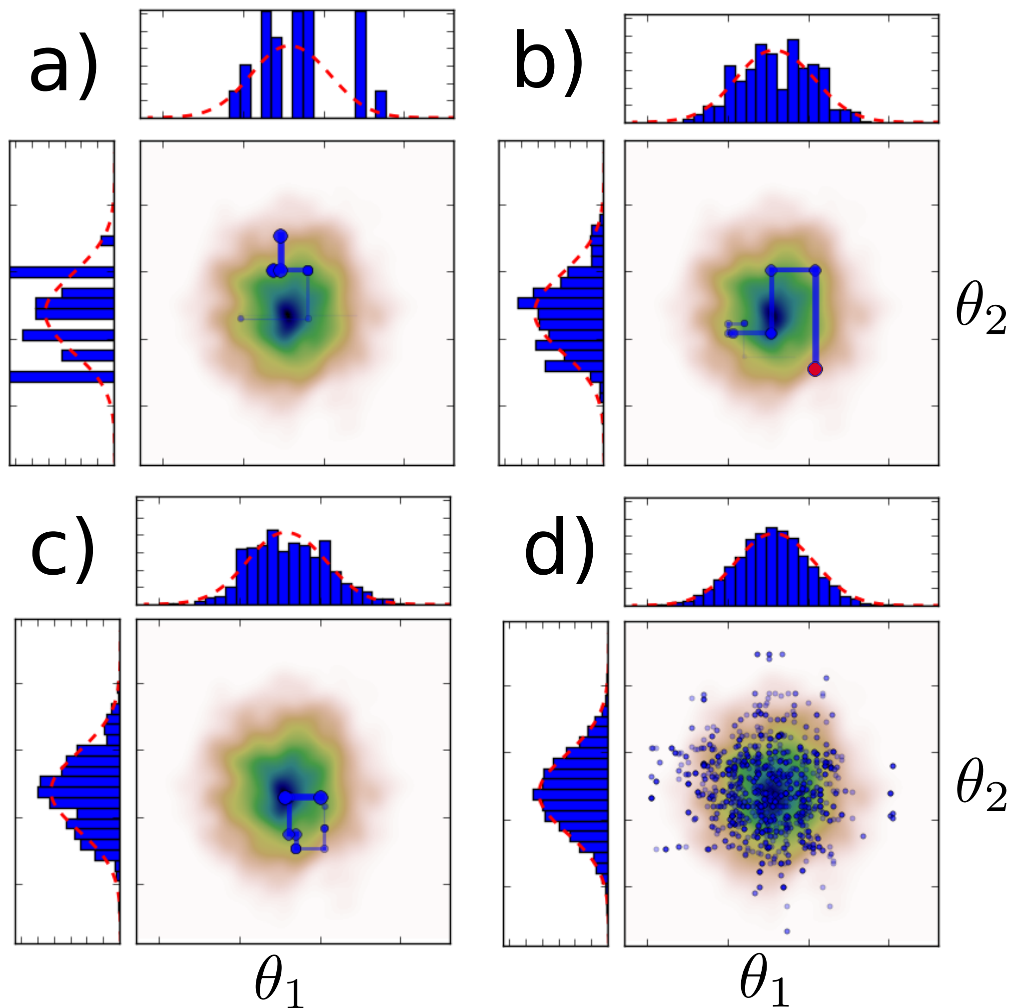


Figure 4.1.: Illustrations of the state of a M-H algorithm and the obtained samples after $k = \{50, 1000, 2000, 35000\}$ iterations when approximating a bivariate independent Gaussian distribution. This target distribution is represented as a color gradient from red (less probable) to blue (more probable), in the center of each figure. The observed samples are summarized in the histograms at the top (θ_1) and left (θ_2) side of each figure, while the target marginal distribution is the red dashed line. Figures a) to c) show the trace of the MCMC as a blue dotted line and figure d) shows the final summary of all the observed samples.

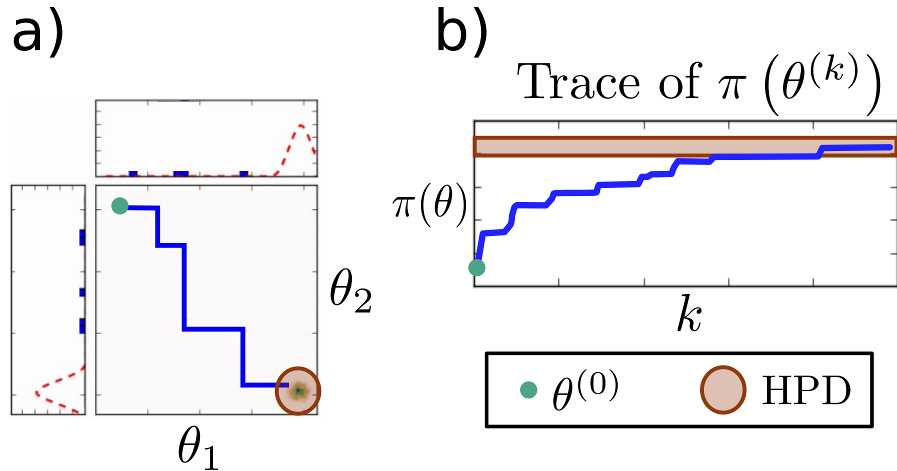


Figure 4.2.: Illustration of the convergence of a MCMC process. The left figure illustrates the initial phase of the sampling of a bivariate Gaussian model with a poor choice of $\theta^{(0)}$. The right figure shows the observed trace of the posterior probability in a similar situation.

The caveat of this approach is that it only holds once the chain has reached its convergence. Using several processors during the burn-in phase would only results in a bigger waste of computing resources and the speedup ensuing from such approach would be significantly limited by the portion of time spent during the burn-in phase.

The final hurdle regarding this practical aspect is the difficulty of assessing the convergence of the chain. Indeed, a chain can reach several plateaux of posterior probability before reaching the HPD, as illustrated by the trace of the chain posterior probability in figure 4.2b). For instance, chains that samples complex and parameter-rich models can get stuck for several thousands of iterations, if not more, in place of the parameter space not included in the HPD of the posterior distribution before moving on. Therefore, solely monitoring the posterior distribution trace gives a limited insight on the progress of the burn-in phase.

For that reason, Brooks and Gelman [19] proposed several measures to assess if a chain, or more accurately, a set of chains has converged. For instance, the multivariate potential scale reduction factor $\hat{R}$ is based on the comparison of the within and pooled variances of all the parameter values observed during multiple MCMC runs. Processing these variances is however a computationally expensive operation that can hardly be applied during the MCMC runs and is thus seldom used afterwards. Therefore, tractable approaches for the identification of the convergence remains an open problem still researched nowadays [45].

Mixing of the chain

The mixing defines how well the MCMC process is evolving in the parameter space and thus determines the quality of the samples derived from it. This property is inherently

linked with the proposal kernel q given that it defines the next considered states for the Markov chain. These proposed states are then kept, or rejected, according to the acceptance probability defined in equation (4.3).

Indeed, a proposal kernel producing *bad* moves will either seldom make the chain progress by proposing new parameter values far from the HPD that are always rejected or, conversely, will make the chain progress frequently by proposing conservative parameter values that slowly explore the parameter space. A *good* proposal kernel, exhibiting a good mixing, will propose new parameter values that fits the form of the posterior distribution and, therefore, the chain will progress frequently by randomly proposing parameter values in the HPD.

This important property can be monitored to diagnose poorly defined proposal kernel. The most used measures for that matter is the integrated autocorrelation time (ACT [39]) that measures the randomness exhibited by a sequence of samples $\mathcal{S} = \left(\theta^{(0)}, .., \theta^{(i)}, .., \theta^{(n)}\right)$. Given a real-valued function $f^{(i)} = f(\theta^{(i)})$, the autocovariance of lag k is defined as

$$\gamma_k = Cov\left(f_n, f_{n+k}\right)$$

with $f_n = \left(f^{(0)}, .., f^{(n)}\right)$ and $f_{n+k} = \left(f^{(k)}, .., f^{(n+k)}\right)$. Based on the autocovariance, the autocorrelation can be readily computed, using the variance $\sigma^2 = \gamma_0$ of f_n, as

$$\rho_k = \frac{\gamma_k}{\sigma^2}. \tag{4.4}$$

When considering the variance $Var_\pi = \tau_n^2/n$ of the expectation estimator $E_\pi\left[f(\theta)\right]$ of the chain having produced $\mathcal{S}$, it can be shown [46] that when $n \to \infty$ then $\tau_n \to \tau$ such that

$$\tau^2 = \sigma^2 \left(1 + 2\sum_{k=1}^{\infty} \rho_k\right).$$

The inner parenthesis term is the ACT and defines how many the observed variance Var_π is impacted by the lag-k autocorrelation, or *measured lack of randomness*, of the sequence $\mathcal{S}$. The ACT is then used to define the Effective Sample Size (ESS) as

$$n_{ess} = \frac{n}{1 + 2\sum_{k=1}^{\infty} \rho_k}.$$

This value gives an insight of the amount of effective samples obtained through the sampling of the Markov chain in function of its observed *randomness*.

More intuitively, assuming that $f(\theta_1)$ would return the observed value of parameter θ_1, then equation (4.4) would give an insight of the correlation between the observed values of θ_1 and the previous values it had k step before. Integrating these values over k gives the overall correlation expressed by the sampled values for θ_1. Therefore, the measured ACT of a randomly drawn sequence should tend to 1 given that no correlations would be observed.

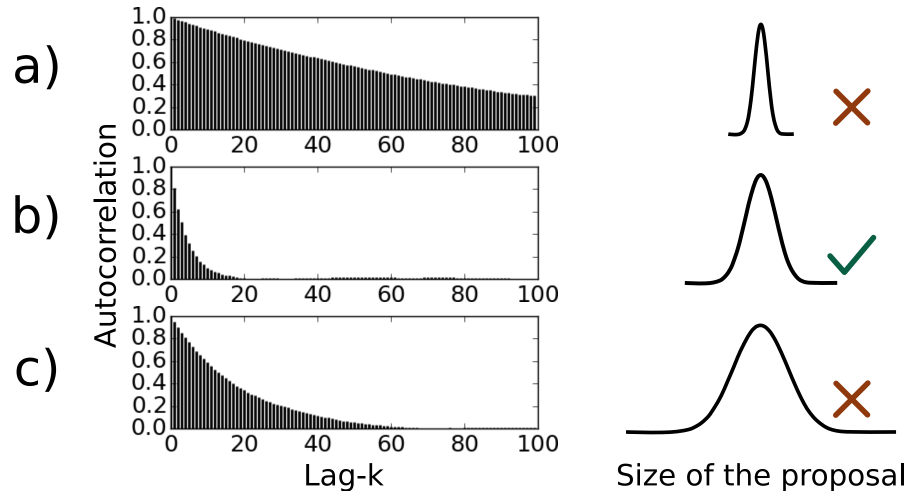


Figure 4.3.: Illustration of the lag-k autocorrelation of three different proposal kernels. Proposals kernels with a small, adequate and large variance with respect to the target posterior density are illustrated in figure a), b) and c), respectively.

Figure 4.3 illustrates the lag-k autocorrelation for three different proposal kernels chosen for the sampling of the bivariate normal model shown (Fig. 4.1). These three proposals are based on univariate Gaussian distributions and therefore define parameter moves as

$$\phi_i^{(k)} = \theta_i^{(k-1)} + \mathcal{N}\left(0, \sigma^2\right) \text{ with } i \in \{1, 2\}\,.$$

The proposal shown in figure 4.3a) is characterised by a small σ^2, and thus small moves that are frequently accepted. Its measured ACT is significant given that each sampled state of the Markov chain is close to the previous one and is thus not *random*. Given that the ACT is far greater than 1, only few out of n sampled states are considered effective.

The opposite case is shown in figure 4.3c) with a proposal having a large σ^2. Most of its proposed parameter values fall out of the HPD and are thus rejected. However, accepted moves are sparsely distributed on the target distribution and are thus more stochastic than the one of the previous kernels. The ACT observed for this proposal is thus smaller than the previous one but, given that the chain stays for several steps in the same state, a significant amount of autocorrelation is still observed.

The final proposal, illustrated in figure 4.3b), is adequately scaled for the task at hand. While moves proposed by this kernel are accepted on average 50% of the time, its measured ACT of 8 remains far from the optimal value of 1. Obtaining $n_{ess} = 1000$ effective sample would then require close to 8000 steps without considering the potential burn-in phase.

4.2. A novel parallel Metropolis-Hastings method

The theoretic aspect of the parallel M-H algorithm specifically designed to obtain an efficient mixing on parameter-rich and complex models is presented in this section. The need for this novel approach, motivated by nowadays evolutionary biology challenges, is justified with respect to the current state-of-the-art of the MCMC method in computational statistics. The two components forming this algorithm are then described, starting with the novel multivariate adaptive proposal that coerces the acceptance rate, balances the mixing along all parameters and exploits their potential correlations. The concept of the second component, a parallel MCMC method, known as *pre-fetching* [18], is introduced and the theoretic concepts used to achieve synergistic performance gains from the coupling of the adaptive proposal with this parallel scheme are explained.

4.2.1. Motivations

In evolutionary biology, MCMC methods have been widely used in the last decades for numerous applications, encompassing phylogenetic reconstructions [106, 77], divergence time analyses [30], molecular evolution [27], comparative methods [35, 13] or population genetics [71, 34]. The complex models used in these applications require costly computational evaluations of their likelihood function and their high-dimensional parameter-space is usually difficult to explore efficiently. The joint effect of the computational effort required for the sampling of these models and the ever growing abundance of novel data has led MCMC samplers to suffer from their inherent sequential nature and their dependency on user defined proposal kernels.

A wide variety of approaches have been proposed to improve the sampling of some of these complex models. These solutions range from avoiding the evaluation of the likelihood function [81, 83] to using sequential Monte Carlo [21, 6] or Hamiltonian Monte Carlo methods [29]. However, these methods suffer from limitations that make them less generally applicable. For instances, some of these methods have to be applied on a model basis, may prove difficult to apply to high-dimensional parameter-spaces or only consider continuous parameters. These limitations are typical of Bayesian inference in phylogenetics. Indeed, the computational complexity of tree reconstruction grows with the length of the sequence data available as well as with the number of taxa, which also defines the amount of parameters and potential phylogenetic trees to consider [33]. Moreover, it is well known that sampling from this discrete space of phylogenetic trees is a particularly difficult task [76, 17].

Current approaches to build phylogenetic trees therefore still heavily rely on MCMC and any enhancement of these methods would be of tremendous use for evolutionary biologists. Several studies focused on the sampling effectiveness (or mixing) of the MCMC process and helped to propose theoretical guidelines to specify the optimal size for proposal kernels based on normal distributions [40, 102]. These guidelines were used to develop the adaptive Metropolis algorithm [51], which aimed at optimally tuning a proposal kernel using the observed empirical covariance of the Markov chain. The adaptive Metropolis method has been further improved by using component-wise scal-

ing [52], adaptively tuning the proposal size to target an optimal acceptance rate [5] or exploiting the parameter covariance [105, 128]. Some of these improvements for univariate proposal kernel have been implemented in the main software to build phylogenetic trees [31, 106, 1].

However, few strategies were explored to improve MCMC efficiency by exploiting parallel computing [47]. While some of these methods present interesting properties, they usually suffer from limitations that hinder their general use on applied problems. For example, prefetching [18] aims at predicting the future states of the Markov chain to pre-process them in parallel. This method is however limited by the difficulty of predicting accurately the path of the Markov chain.

The novel parallel algorithm presented here aims to fill this existing need for a generally applicable parallel MCMC method that would improve the sampling of complex parameter spaces such as the discrete space of phylogenetic trees.

4.2.2. `EMPIR`: an efficient multivariate adaptive proposal

The presented M-H algorithm uses its own Efficient and balanced Multivariate adaPtIve pRoposal kernel, `EMPIR`, specifically designed to maintain a low computational overhead while exploiting the potential correlations between parameters in complex and parameter-rich models.

The original adaptive method presented by Haario et al. [51] defines the proposal as $\phi \sim \mathcal{N}(\theta, \lambda\Sigma)$ with λ being the optimal scaling factor defined by Gelman *et al.* [40] and Σ the observed empirical covariance of the Markov chain. Given the sample $\theta^{(k)}$ observed at iteration k of the MCMC process, the parameter mean $\bar{\theta}$ and covariance Σ are updated using the following recursive formula,

$$
\begin{aligned}
\bar{\theta}^{(k+1)} &= \frac{k-1}{t}\bar{\theta}^{(k)} + \frac{1}{k}\theta^{(k)} \\
\Sigma^{(k+1)} &= \frac{k-1}{t}\Sigma^{(k)} + \frac{1}{k}\Big[k \cdot \Big(\bar{\theta}^{(k-1)} \otimes \bar{\theta}^{(k-1)}\Big) - \\
&\qquad\qquad (k+1) \cdot \Big(\bar{\theta}^{(k)} \otimes \bar{\theta}^{(k)}\Big) + \Big(\theta^{(k)} \otimes \theta^{(k)}\Big)\Big].
\end{aligned}
$$

While being exact, this method suffers from an expensive computational cost due to the three outer products required per iteration. Moreover, effects from parameter values, that are not representative of the target distribution $\pi(\theta)$, observed during the burn-in phase have the tendency to require several iterations to be attenuated.

In order to avoid these caveats, `EMPIR` approximates the covariance matrix Σ using a more evolved approach proposed by Andrieu and Thoms [5]. Being based on stochastic approximation, this approach is less perturbed by the burn-in phase, as illustrated in figure 4.4, and optimizes the mixing by coercing the acceptance rate which is more robust than relying on λ. Using stochastic approximation, the global scaling factor λ,

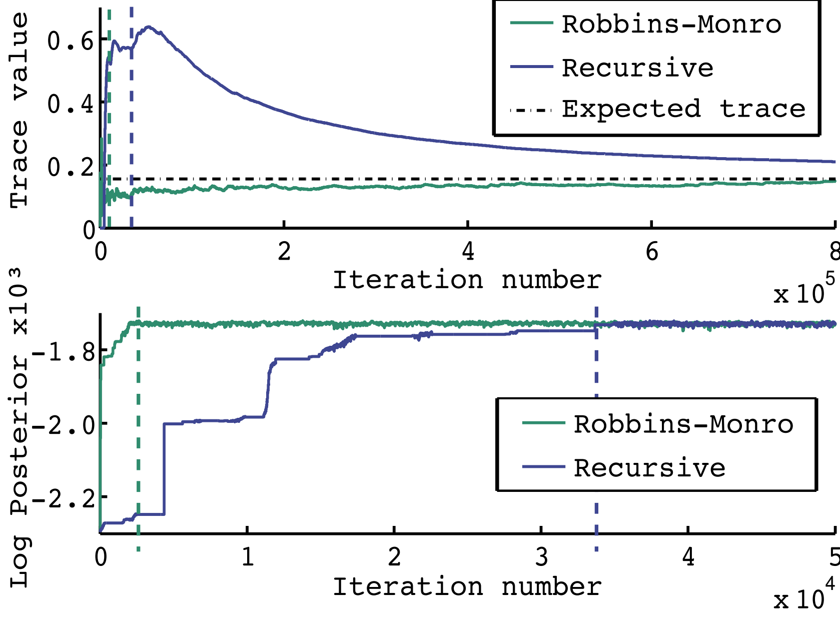


Figure 4.4.: Illustration of the advantage of the stochastic approximation approach (*Robbins-Monro*) over the original adaptive method (*Recursive*). The top figure shows the aproximated value during a MCMC run of an adaptively learned parameter, e.g. the global scaling λ. The bottom figure shows the posterior probability during the same MCMC run.

the parameter mean $\bar{\theta}$ and covariance Σ are updated as follows

$$\begin{aligned} \bar{\theta}^{(k+1)} &= \bar{\theta}^{(k)} + \gamma^{(k)} \cdot \left(\theta^{(k)} - \bar{\theta}^{(k)}\right) \\ \Sigma^{(k+1)} &= \Sigma^{(k)} + \gamma^{(k)} \cdot \left[\left(\theta^{(k)} - \bar{\theta}^{(k)}\right) \otimes \left(\theta^{(k)} - \bar{\theta}^{(k)}\right) - \Sigma^{(k)}\right] \\ \log\left(\lambda^{(k+1)}\right) &= \log\left(\lambda^{(k)}\right) + \gamma^{(k)}\left(\bar{\alpha}^{(k)} - \alpha_\star\right) \end{aligned} \tag{4.5}$$

with $\alpha_\star$ defining the target acceptance rate that optimizes the mixing [102]. The step size γ impacts the convergence of the approximation process and must have the following two properties [101]: $\sum_{k=0}^{\infty} \gamma_k = \infty$ and $\sum_{k=0}^{\infty} \gamma_k^2 < \infty$.

Coercing the global scaling factor λ guarantees that the overall proposal acceptance rate is optimal by taking advantage of the information made available by the MCMC process. However, it does not ensure a balanced mixing over all parameters. Component-wise scaling is able to solve this problem at the significant cost of d additional likelihood evaluations [52, 5]. The new component-wise or local scaling factors $\Lambda = diag(\lambda_1, ..., \lambda_d)$, estimated separately for each parameter, can then be used to scale the proposal over each dimension (corresponding to a parameter) as follows:

$$\phi \sim \mathcal{N}(\theta, \Lambda^{1/2}\Sigma\Lambda^{1/2}) \tag{4.6}$$

This approach guarantees a more balanced mixing over all parameters while inducing a significant increase in computational cost. A second drawback is the loss of control on the global acceptance rate. Indeed, independent parameters would lead to a global acceptance rate defined as $\alpha = \prod_{i=1}^{d} \alpha_i$. However, in the more frequent case of correlated parameters, the relation between local acceptance rates α_i and the global acceptance rate α remains undefined.

In order to efficiently balance the mixing along all parameters, `EMPIR` combines both local and global scaling variants. This proposal maintains the coercion of the global acceptance rate by using the global scaling λ, which is updated at each iteration. The costly component-wise updates are made periodically and the normalized scaling factors $w_i = \lambda_i / \sum_{k=1}^{d} \lambda_k$ form the local scaling matrix $W = diag(w_1, ..., w_d)$. The proposal then uses the global scaling to target the overall size of the move while the normalized local scaling ensures a balanced mixing over all directions.

$$\phi \sim \mathcal{N}(\theta, \lambda W^{1/2} \Sigma W^{1/2}) \tag{4.7}$$

This method is further enhanced by considering the correlations observed between parameters θ as proposed by Gilks *et al.* [42]. These correlations are exploited by using the geometric interpretation of the multivariate normal distribution. Since this distribution belongs to the family of elliptical distributions, it can be expressed by the directions of the principal axes of the ellipsoids. The spectral decomposition of the covariance matrix

$$\Sigma = QEQ' = QE^{1/2}(QE^{1/2})'$$

gives these directions as the eigenvectors $Q = (V_1, .., V_d)$ and their scaling as the eigenvalues $E = diag(e_1, .., e_d)$. Samples can then be obtained from the distribution

$$X \sim \mathcal{N}(\mu, \Sigma) \equiv \mu + QE^{1/2}\mathcal{N}(0, I)$$

The component-wise scaling factors $\tilde{\lambda}_i$ are used in this case to scale the size of the moves along the direction of the eigenvector V_i and the scaled eigenvectors $(\lambda \tilde{W} E)^{1/2}$ can be used to build the following proposal

$$\phi \sim \theta + Q(\lambda \tilde{W} E)^{1/2} \mathcal{N}(0, I) \tag{4.8}$$

This method however comes at an additional computational cost that grows exponentially with the number of dimensions d. Indeed, the covariance matrix is costly to learn since it requires linear algebra functions with complexity $\mathcal{O}(d^2)$. In addition, the generation of random moves based on multivariate distributions requires the Cholesky or eigendecomposition of the scaled covariance matrix at a computational cost of $\mathcal{O}(d^3)$. Moreover, this decomposition must be updated whenever Σ and Λ change.

4.2.3. `P-EMPIR`: Securing an optimal pre-fetching with `EMPIR`

Markov chains are an inherently sequential process due to the dependencies between states in two subsequent iterations of a chain. Obtaining speed improvements by using

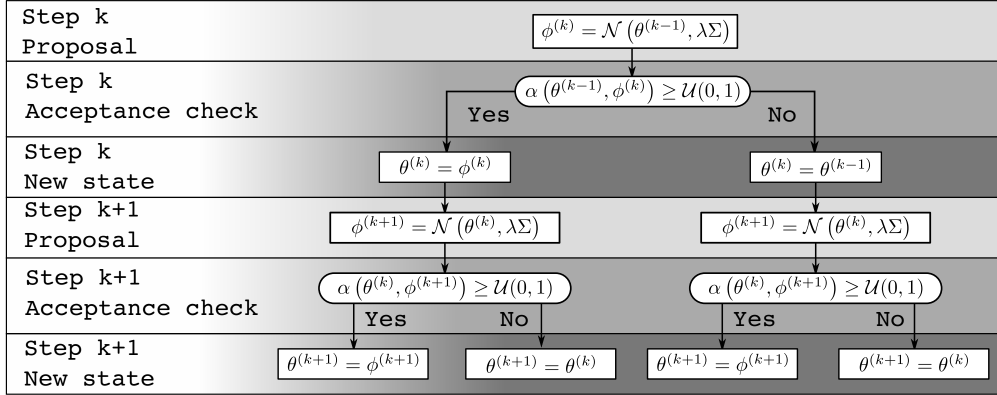


Figure 4.5.: Markov chain decision tree of height $h = 3$

parallel computing techniques is therefore an important challenge. Pre-fetching [18] overcomes this limitation by pre-processing the possible paths that the Markov chain could take during a set of iterations.

The future path of a MCMC can be represented as a decision tree (Fig. 4.5). Given that the chain at the beginning of step k is in state $\theta^{(k-1)}$, a new state $\phi^{(k)}$ is proposed. This new state is then accepted or rejected with probability $\alpha(\theta^{(k-1)}, \phi^{(k)})$. Branches of the tree represent these two possible paths of the MCMC process. Each state, or node, leads to two subtrees corresponding to either an acceptance or a rejection.

The likelihood computation of the future possible states ϕ can then be distributed over multiple processors. Given that several samples are generated during the same amount of time that it previously took for one, the M-H algorithm is significantly sped up. However, given the tree structure of the possible paths of the Markov chain, an exhaustive approach to get $D = h$ samples per iteration would require the computation of the $2^h - 1$ possible states of the chain and thus, as much processors. This strategy scales poorly since the number of wasted likelihood computations corresponding to unused states grows exponentially with D.

Various strategies [123] aiming at determining the most probable path in the decision tree have been proposed to improve the scaling of this method. One of these strategies uses the mean observed acceptance rate of the chain $\bar{\alpha}$ as a predictor for the most probable paths. The efficiency of this method is estimated by a performance model that depends on the acceptance rate and the number P of available processors:

$$E(\alpha, P) = E^1(\alpha) \cdot D(\alpha, P) \tag{4.9}$$

where $E^1(\alpha)$ defines the mixing efficiency for an i.i.d normal distributed proposal with $d \to \infty$ [104] and $D(\alpha, P)$ is the expected number of draws per pre-fetching iteration [123].

The expected number of draws $D(\alpha, P)$ is equal to P and thus optimal for $\alpha = 0$ or $\alpha = 1$. With such α, pre-fetching is indeed able to exactly predict the path of the

Markov chain by considering moves as being always rejected, respectively accepted. Such a prediction would be represented by the right most, respectively left most, branch in the decision tree (Fig. 4.5). The expected number of draws $D(\alpha, n_p)$ reduces as α approaches 0.5, since predictions are less accurate. When α reaches 0.5, any path in the decision tree is equally probable and thus pre-fetching is back to using the inefficient exhaustive approach of computing all the possible paths.

On the other hand, the mixing efficiency of a proposal $E^1(\alpha)$ peaks at an optimal α value of 0.234 with $d \to \infty$ [104]. Thereof, the optimal acceptance rate $\alpha_\star$ for a given number of processors derived from equation (4.9) expresses the optimal trade-off between an efficient mixing and accurate predictions for pre-fetching. In other words, the efficiency of both methods is then controlled by the average acceptance rate α of the proposals and is optimal for $\alpha_\star$.

Therefore, the coupling of pre-fetching with `EMPIR` is done by coercing the acceptance rate to the optimal value $\alpha_\star$ through the global scaling factor λ. The resulting coupling, later referred as `P-EMPIR`, ensures that the acceptance rate of the MCMC process is quickly reaching the vicinity of the target acceptance rate $\alpha_\star$. Once this target value is reached, `P-EMPIR` optimizes the efficiency of the pre-fetching method by using the optimal size of the proposal kernel; optimal size defined in function of the number of available processors and regardless of the model.

In return of providing these optimal conditions for the pre-fetching method, `EMPIR` also benefits from this coupling. During a pre-fetching iteration, only a certain number of predicted states are retained, while the remaining unused ones are discarded. However, `EMPIR` takes advantage of these wasted likelihood evaluations. During one iteration, all the acceptance rates α estimated to predict the chain path in the decision tree offer usable information that can help the updating of λ. The advantage is that the learning process becomes more accurate and adapts more quickly to the parameter space, because λ is adapted more frequently. This results in a better sampling efficiency and reduces the length of the burn-in phase. These improvements apply equally well, if not better, to the local scaling factors Λ by processing the d unidimensional moves in parallel and thus offering a nearly ideal speedup of this costly operation.

4.3. Implementation of `EMPIR` and `P-EMPIR` in `HOGAN`

While the methods presented in the previous section are in theory sound, in practice they require a careful implementation in order to reveal their full potential. Indeed, `EMPIR`'s potential to enhance the mixing of the M-H algorithm is hindered by its computational cost that exponentially grows with the number of parameters. Therefore, this trade-off must be adequately dealt with to extract the most of `EMPIR`.

Similarly, `P-EMPIR` relies on parallel evaluations of multiple likelihoods to accelerate the M-H algorithm. However, in the context of `HOGAN`, likelihood evaluations have varying computational costs and therefore, distributing these evaluations on P processors may cause load balancing issues.

Finally, the MCMC method are known to have difficulty to sample multimodal dis-

tributions. Additional enhancements are thus required for conducting efficiently the sampling of such distributions. These three practical issues are considered in the implementation of EMPIR and P-EMPIR in HOGAN. The strategies and enhancements required to deal with these problems are detailed in this section.

4.3.1. Optimising the efficiency of EMPIR

The efficiency of EMPIR depends on the trade-off between the performance gain coming from multivariate proposals with support for correlated parameters and the overhead caused by these methods that comes from the eigendecomposition and the Cholesky factorization of the covariance matrix Σ. Three strategies used to enforce this trade-off are here described.

The first one uses several proposal kernels applied to subsets of parameters in order to reduce the exponential overhead coming from these complex linear algebra operations. The second one monitors the adaptive process of the proposal kernels and ends it when the values of λ, Λ and Σ are detected as converging. The final one makes EMPIR evolve from a simple and costless adaptive kernel to the more evolved and expensive one defined in equation (4.8) in function of the approximated covariance matrix Σ.

Using smaller blocks

In theory, updating all parameters during one step of the M-H algorithm with an adequate proposal kernel offers the best possible mixing [39]. However, in practice, proposal kernels moving large amount of parameters affect negatively the efficiency and the stability of EMPIR. Indeed, its efficiency was already identified to depend on the computational cost of linear algebraic operations that exponentially grows with the number of parameters. In addition to this unpleasant effect, the stability of adaptive proposals suffers from the difficulty to accurately approximate high-dimensional covariance matrices.

A natural solution addressing these problems is to consider the use of blocks containing small subsets of the d parameters. Each of these blocks would propose moves on their parameters according to a proposal kernel dedicated to the block. The resulting M-H algorithm would thus be identical to the one employed for univariate proposal kernels that was described in section 4.1. However, to adequately benefit from this readily applicable solution, several considerations on the composition and size of the blocks have to be taken into account in the context of P-EMPIR and HOGAN.

Indeed, the target acceptance rate $\alpha_\star$ defining the efficiency of P-EMPIR decreases as the number of processors increases (Fig. 4.6). However, a small $\alpha_\star$ tends to produce large and bold moves that are difficult to tune reliably when few parameters are updated. Fortunately, increasing the number of parameters per block tends to limit the boldness of these moves. Indeed, assuming that the parameters in a block are independent, the optimal acceptance rate of a block of parameter identified by i is given by

$$\alpha_\star^i \approx \prod_{j=1}^{d^{(i)}} \alpha_j \tag{4.10}$$

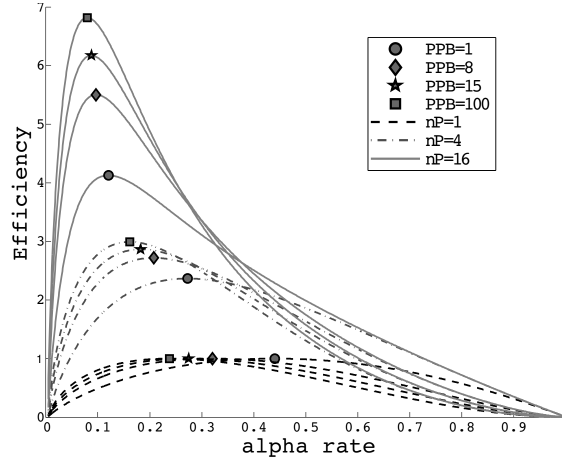


Figure 4.6.: Alpha's efficiency, representing the average expected samples per prefetching iteration, in function of the number of parameters per block and processors

where $d^{(i)}$ is the size of the block i and α_j stands for the acceptance rate of the univariate moves along direction j. Therefore, parameter blocks should be big enough to benefit from the previously identified stability induced by large blocks.

Moreover, the creation of blocks of parameters, or *blocking* [103], is often used to decompose the likelihood in smaller sub-likelihoods, if the model permits. These sub-likelihoods define conditional probabilities that compose the full likelihood and an update of their assigned parameters enables their sampling at a lower computational cost. This strategy, called *belief propagation*, already mentioned in chapter 2, is widely used in graphical models [95] and is to conditional probability what PLUs are to computational tasks. Therefore, blocking directly benefits from PLUs due to the reduced computational cost induced by considering moves on small parameter blocks.

All these considerations affecting the efficiency of the resulting sampler must be taken into account to define the optimal size of a parameter block. This daunting task is therefore either avoided by solely considering univariate proposal kernels or delegated to the end user. For this last option, general guidelines, obtained from empiric experiments with `P-EMPIR`, advises users to define block size in the range of 10 to 50 parameters.

Given that such blocks of parameters are not large enough to consider that their size $d^{(i)}$ tends to infinity, the pre-fetching performance model defined in equation (4.9) has to be slightly adapted. A small correction on $E^1(\alpha^{(i)})$ is applied to take into account the block size d_i as follows

$$E^1(\alpha^{(i)}) \propto \alpha^{(i)} \cdot \left[\varphi^{-1}\left(\frac{\alpha_i}{2}\right)\right]^{\rho}$$

with $\rho = 2 - 1.1e^{-d_i/10}$ and where φ stands for the standard normal distribution. This correction is based on empirical observations made by Roberts *et al.* [104] and the one made during the upcoming experiments with `P-EMPIR`.

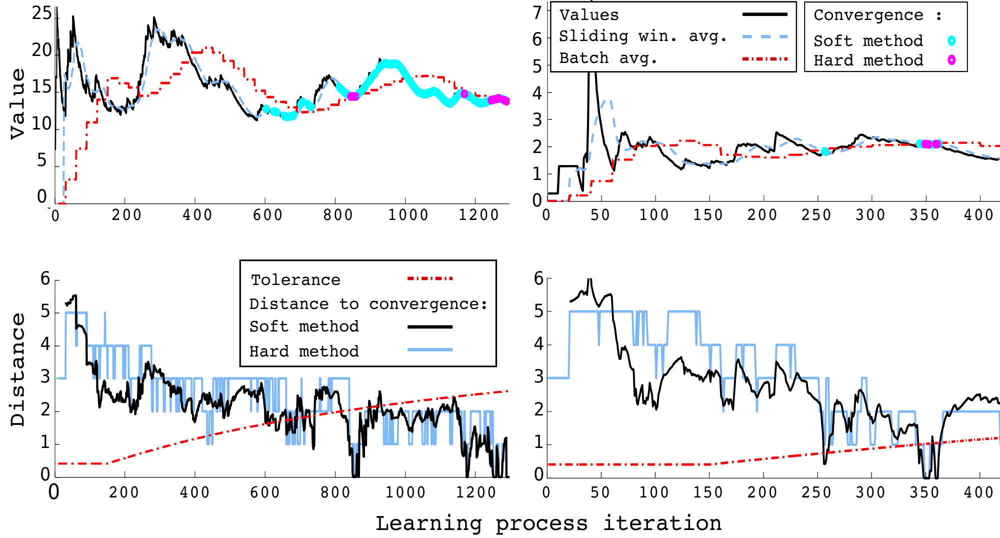


Figure 4.7.: Early convergence detection. Upper figures show examples of learned variables values and their respective averages while dots represent the detection of a soft or hard convergence. Distance to convergence are shown with plain line in lower figures while the red dotted line depicts the tolerance threshold $\epsilon^{(k)}$ of the soft method and its relaxation.

Additionally to the block size, one last important aspect regarding the definition of blocks remains unaddressed: the choice of parameters that must be grouped together. This choice is an even bigger challenge than the one caused by the block size. Indeed, the mixing of the MCMC sampler depends directly from this choice given that it determines, by choosing which parameters to group in blocks, the parameter correlations that can be exploited. Furthermore, the performance gains resulting from the PLUs, or similar artifices, is directly defined by the grouping of parameters in chapter 3. While this choice is also usually delegated to the user or defined empirically on a model-specific basis, a promising concept is presented in the last section of this chapter.

Convergence detection

The stochastic approximation scheme used to learn the values of λ, Λ and Σ (Eq. (4.5)) is theoretically guaranteed to converge as the number of iterations tends to infinity. However, continuously refining the approximation of these values has a significant computational cost that may not be worth paying once a certain level of accuracy is reached. Indeed, in order to achieve an effective proposal kernel, these variables must adequately approximate the form of the posterior distribution but are not required to exactly match it. Therefore, in `EMPIR`, the convergence of these variables is monitored and their learning phase is stopped once a decent level of stability is reached such as to compromise between their accuracy and the computational effort spent for their approximation.

Detecting the convergence of these variables without increasing the computational cost of the algorithm is challenging given that their approximated value through time may be highly volatile and sometimes nearly periodic (Fig. 4.7). Therefore the convergence detection at iteration k is based on multiple measures of stability $\upsilon_i^{(k)}$ on different temporal scales such as the current value, the last l values (sliding window) or the batch of average values (fixed batch windows). These measures $\upsilon_i^{(k)}$ represent, for instance, the distance between the current value and the average of the sliding window or the relative standard deviation of the batch windows.

Using these measures, the hard convergence of the learning phase can be defined as

$$\bigcap_{i=1}^{m} (\upsilon_i^{(k)} < \tau_i),$$

with τ_i being pre-defined convergence thresholds, each associated to one measure $\upsilon_i^{(k)}$. However, this condition of convergence may be difficult to achieve given that all the separate criteria must be fulfilled at the same time. Therefore a softer approach was designed to accept a small divergence ϵ on the set of thresholds τ_i such that

$$\left[\sum_{i=1}^{m} \max\left(\frac{\upsilon_i^{(k)} - \tau_i}{\tau_i}, 0\right)\right] < \epsilon.$$

In addition of being more flexible, this approach enables the relaxation of ϵ as the number of iterations k increases and thus offers an interesting leverage to loosen the convergence criterion. This relaxation defines ϵ as

$$\epsilon = \beta_1 \cdot \beta_2 \cdot \epsilon_0,$$

where β_1 is the relaxation factor local to each of the learned variables and β_2 is the global relaxation factor that considers the overall state of the learning phase.

The local relaxation factor β_1 ensures that after a certain amount of time spent learning a given variable, ϵ is gradually relaxed such as to facilitate the convergence detection for volatile variables. Therefore, β_1 is defined as

$$\beta_1 = \max\left(1, \frac{k}{K_{lim}}\right)^{\Gamma_1},$$

where K_{lim} defines a threshold of iterations after which the local relaxation should occur and Γ_1 tunes the rate of relaxation increase through time.

The global relaxation factor β_2 avoids that the overall adaptive process gets stuck in the learning phase because of a small amount of non-converging variables. Assuming that a proportion ρ of the approximated variables has been detected as having converged, the factor β_2 is given as

$$\beta_2 = \max\left(1, \frac{\rho}{\rho_{lim}}\right)^{\Gamma_2},$$

where ρ_{lim} defines the threshold of proportions required to activate the global relaxation and Γ_2 serves the same purpose as Γ_1.

This flexible approach requires a large amount of constant to tune the behaviour of the convergence detection. Therefore, several pre-settings corresponding to different trade-offs between the approximation accuracy and the convergence rate are readily available in `HOGAN`. Therefore, the setting of these constants can be customised or readily imported by simply specifying a required rate of convergence such as *fast*, *default* or *slow*.

Automated choice of the most appropriate proposal kernel for `EMPIR`

The most evolved proposal kernels defined by equation (4.7) and (4.8) have the advantage of exploiting the observed parameter correlations at the expense of costly linear algebra operations. Unfortunately, the definition of these proposals is such that these operations are applied regardless of the amount of correlations observed. Even when insignificant levels of correlations are observed, this expensive computational cost is paid.

Therefore, `EMPIR` chooses intelligently the proposal kernels the most appropriate given the observed covariance matrix Σ. Three different proposals are available for `EMPIR` to chose. The first one is a simplification of equation (4.7) based on the assumption that parameters are uncorrelated such that

$$\phi \sim \mathcal{N}(\theta, \lambda W^{1/2}\tilde{\Sigma}W^{1/2}), \tag{4.11}$$

with $\tilde{\Sigma} = diag\left(\sigma_1^2, ..., \sigma_d^2\right)$ and thus,

$$\phi_i \sim \mathcal{N}(\theta_i, \lambda w_i \tilde{\sigma_i}).$$

The second and third one are defined by equations (4.7) and (4.8).

The starting proposal chosen by `EMPIR` is always the first one. The variables λ, Λ and Σ are approximated until a first convergence of their learning process is detected under an *extra fast* setting. Then the amount of correlations expressed in Σ is assessed and whenever it exceeds a given threshold[2] the proposal is switched to either the second or the third one. The approximated variables are then subject to a final learning phase that is stopped when the convergence is detected according to the initial setting specified for the convergence rate.

Therefore, three schemes can be chosen for `EMPIR`:

- `STD` that starts with the first proposal and does not evolve.
- `MIXED` that starts with the first proposal and may evolve into the second proposal kernel (Eq. (4.7)).
- `PCA` that starts with first proposal and may evolve into the third proposal kernel (Eq. (4.8)).

The overall best performing scheme was identified to be the `PCA`, as later demonstrated, and is thus the default scheme used for `EMPIR`.

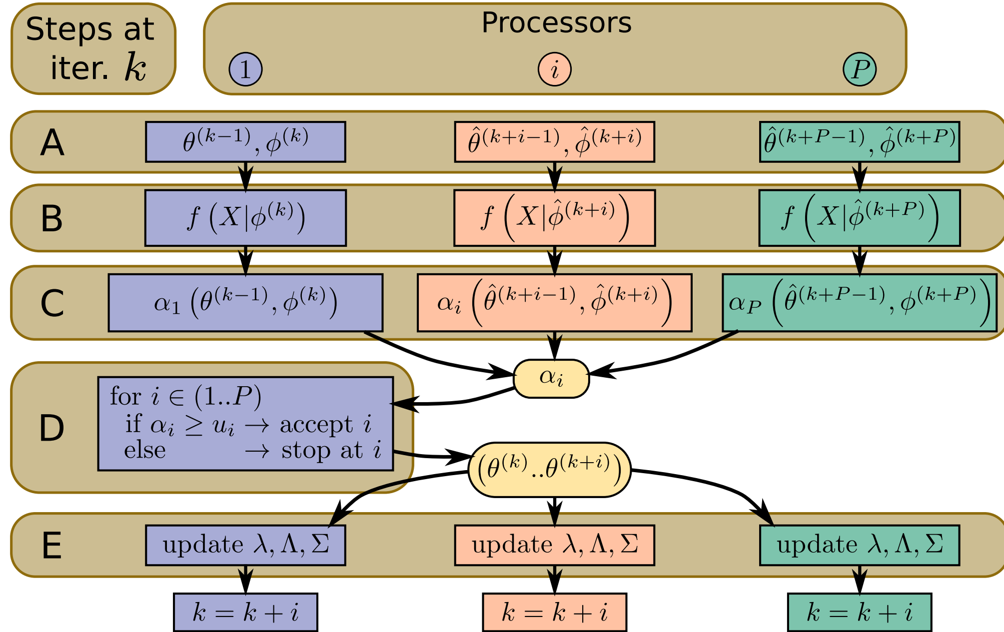


Figure 4.8.: Simplified illustration of the major operations during iteration k of `P-EMPIR`: A) prediction of the Markov chain path, B) parallel likelihood evaluations, C) local acceptance probability, D) prediction verification on the main processor, and E) adaptive process. After the prediction validation, the main processor broadcasts the first ith correctly predicted moves. Parameters identified by a hat, $\hat{\cdot}$, identify predicted moves.

4.3.2. Managing unbalanced likelihood evaluations in P-EMPIR

The key operations applied on P processors during an iteration of P-EMPIR are illustrated in figure 4.8. Each processor starts by predicting the path of the Markov chain. This step is straightforward given that the $\alpha_\star$ derived from the efficiency equation (4.9) leads to a prediction where all the samples are rejected. Based on this prediction, each processor evaluates the likelihood and the acceptance probability for its respective predicted parameters ϕ. The main process gathers all these acceptance probabilities and assesses which predictions are proved correct. These correctly predicted moves $(\theta^{(k)}, .., \theta^{(k+i)})$ are then shared with all the other processors that then update the adaptive proposals accordingly. Wrongly predicted moves $(\theta^{(k+i+1)}, .., \theta^{(k+P)})$ are thrown away.

Given that the most expensive step is the likelihood evaluations for complex models, this approach could, in theory, lead to a linear speedup with respect to P. In reality, the likelihood evaluations are indeed the time dominant step, however their respective computational costs are usually highly unbalanced. Two issues are at the root of this problem and are addressed in this section.

The first reason are the bounds $l_i < \theta_i < u_i$ imposed on the parameters. When a proposal kernel suggests a move that results by parameter being out of their boundaries, the likelihood has by default a null value and thus requires no computation. To avoid this problem. parameters are usually reflected inside the parameter space. However, reflecting parameters suggested by multivariate proposals that have been rotated (PCA) is not a trivial task.

The second reason is caused by the use of parameter blocks. While this approach enables PLUs to reduces the cost of likelihood evaluations induced by moves on these parameter blocks, it also creates fluctuating computational costs for each parameter block. For instance, on the task of inferring a large phylogeny, PLUs induced by the change of evolutionary model parameters are two orders of magnitude slower than PLUs induced by the update of branch lengths.

Reflection of bounded parameters

The method used to reflect the parameters inside their bounds is detailed in algorithm 8. The main concept is to reflect the vector $\vec{v}$ going from θ to ϕ in its opposite direction, for the yet untravelled distance, once the first boundary is encountered as illustrated in figure 4.9. This operation is repeated until the initial length of $\vec{v}$ has been travelled. Using always the first encountered boundary along $\vec{v}$ is required to ensure that the final parameter values end inside their boundaries. Moreover, this ordering also guarantees that this reflection is reversible given that the reverse move from the reflected ϕ along the vector $-\vec{v}$ encounters the same boundaries but in reverse order.

Reflections must however be tightly controlled given that they may wrongly mislead the adaptive process of the proposal kernels. Indeed, if a tightly bounded parameter always leads to moves that are accepted with a probability $\alpha > \alpha_\star$, then the natural tendency of the adaptive process will be to increase the scaling factor λ to produce bolder

[2] This threshold is arbitrarily chosen to but can be customised.

Algorithm 8 Reflection for multivariate normal proposals

```
// Initial proposal with constraints:  l ≤ θ ≤ u
θ' = θ
v⃗ = N(0, λΣ)
φ' = θ' + v⃗
repeat
  // Find closest intersection with a bound
  I = ∅
  d = ∞
  for all θ'_i ∈ θ' do
    if θ'_i > u_i or θ'_i < l_i then
      // Define the distance between θ' and the b along v⃗
      if θ'_i > u_i then b = u_i else b = l_i end if
      b = ‖v⃗‖ · |(b − θ'_i) / (φ'_i − θ'_i)|
      // Keep the smallest distance
      if b < d then
        d = b
        I = {i}
      end if
    end if
  end for
  // Move θ' to the intersection and reflect φ'
  if I ≠ ∅ then
    u⃗ = v⃗/‖v⃗‖
    w⃗ = d · u⃗
    θ' = θ' + w⃗
    v⃗ = −1 · (‖v⃗‖ − ‖w⃗‖) · u⃗
    φ' = θ' + v⃗
  end if
until I = ∅
return  φ = φ'
```

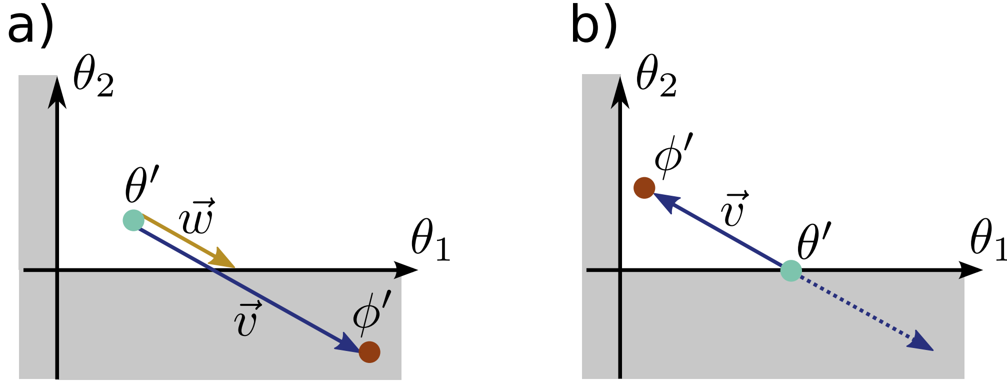


Figure 4.9.: Illustration of a simple reflection. The left figure shows the initial proposed parameter values, while the right figure shows the resulting effect of the reflection. Greyed zones represents unsupported part of the parameter space.

moves that should be less frequently accepted and thus reduce α. Unfortunately, given that the proposals are reflected the moves will not modify the observed α. This scenario can cause two unpleasant effects.

The first is the divergence of the scaling λ that may cause overflows and create instability in the parameter block investigated. The second is closely related given that the number of reflections required to bring back the parameter in its bounds increases with the divergence of λ. This form of ping-pong of the parameter between its boundaries may require a large number of iterations in algorithm 8 and thus a significant computational cost that could negatively impact the performance of the `P-EMPIR`.

In order to deal with these issues, `P-EMPIR` uses a counter that is incremented when more that r reflections are monitored on the same parameter during one call of algorithm 8. These events are signalled to the adaptive process that then tunes down the scaling factor λ. Whenever this scheme is repeated many times, the conflicting parameter is decoupled from its parameter block and its adaptive process is stopped.

Selecting blocks of parameters

Multiples strategies can be used for the selection of the parameter block to update at each iteration of the M-H algorithm: for instance, blocks can be sequentially or randomly selected. The MCMC process converges to the target distribution $\pi(\theta)$ as long as the selection of blocks does not depend on the current parameter values θ or their posterior probability $\pi(\theta)$ [39].

Assuming that a random selection strategy would be used, for each iteration of `P-EMPIR` processors would select randomly a block and then propose a move on it. This approach results in an unbalanced computational load on the P processors given that each parameter block would induce a different PLU. Therefore, parameter blocks producing high variances of likelihood computational times would result with some pro-

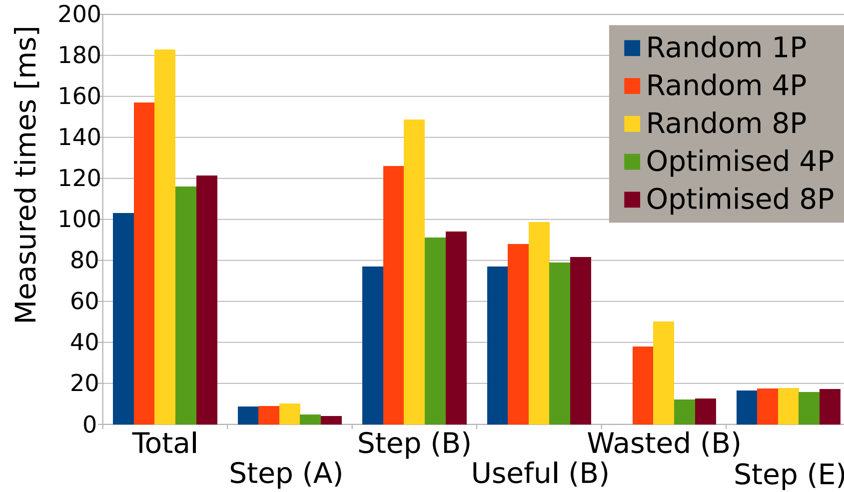


Figure 4.10.: Illustration of the distribution of the computational efforts during an iteration of `P-EMPIR` with the default **random** block selection and the **optimised** one. The steps correspond to the one of figure 4.8 with *Step (B)* being the sum of the average times *Useful (B)* and *Wasted (B)*. The first represents the effective time spent evaluating likelihoods and the seecond one the time spent waiting on other processors to finish computing their likelihoods.

cessors having to wait on others to finish their computation, representing thus a waste of computational resources. This performance loss is shown in figure 4.10 with illustrative measures using the *random* strategy.

A tempting solution to this problem would be to randomly select a unique block that would be used by each processor. Given that all processors would compute the same PLU with different values, the load would be perfectly balanced with this approach that would be equivalent to P consecutive proposed moves on this unique block. Furthermore, the efficiency equation (4.9) defines that the target $\alpha_\star$ decreases as P increases. A gross approximation of this trend gives that P processors would require on average P consecutive proposed on a block to get an acceptance. Therefore, in addition of levelling the computational cost this approach would accept, on average, a proposed sample per `P-EMPIR` iteration.

While this is true, naively applying such an approach would bias the MCMC sampler by proposing more or less moves according to the current parameters posterior probability. Indeed, considering parameters θ' and θ'' with $\pi(\theta') \gg \pi(\theta'')$, on which P moves from the same proposal kernels would be suggested, then, on average, the moves from θ'' would be more frequently accepted than the ones from θ' given that θ'' has more room for improvement. Since pre-fetching predictions of the Markov chain path are based on a scenario where each sample is rejected, the average number of correct predictions i' from θ' would be higher than the i'' from θ''. Given that the wrongly predicted moves,

occurring after the i correctly predicted, are thrown away: the parameters θ' having a better posterior probability would be more frequently sampled given that $i' > i''$.

This dependency between the posterior probability of the current parameter values and their sampling frequency can be readily negated by ensuring that proposals are always applied subsequently as many times as required to get P correctly predicted moves. Assuming that during the first iteration $P - m$ moves are correctly predicted, then m moves should still be proposed for the current block. For that task, m processors suffice and thus two possibilities are conceivable for the next iteration: either $P - m$ processors remain unused or these processors begin to compute the next P moves dedicated to the next randomly chosen parameter blocks.

This second strategy, referred as *optimised*, is the one implemented in `P-EMPIR` with a default value of $(1.2 \cdot P)$ consecutive moves per block. Even if two different parameter blocks are frequently sampled at the same time, this approach greatly reduces the variance of the likelihood computational cost dispatched over the P processors. The reduction of the computational resources wasted by waiting for the completion of the likelihood evaluation of other processors is illustrated in figure 4.10.

4.3.3. Dealing with multimodal posterior distributions

The sampling of multimodal posterior distribution is well known to be problematic for the M-H algorithm [39]. However, this problem can be easily tackled with the use of the *Metropolis-coupled Markov chain Monte Carlo* strategy (MC^3 [41]) that is no stranger to computational biology [139, 4]. This method is directly related to the *Parallel Tempering* [94] and considers the tempering of the target density $\pi(\theta)^{\beta_0}$ by defining m auxiliary distributions with density defined as $\pi(\theta)^{\beta_i}$ with *heats* $\beta_i \in [0, 1]$ and $1 = \beta_0 > \beta_1 > .. > \beta_m > 0$. Given that these auxiliary distributions are *flattened* versions of the original density $\pi(\theta)$, the modes are attenuated and thus moves across them should be facilitated.

The concept is then to sample in parallel the original as well as the auxiliary densities using MCMC samplers and to periodically exchange the current state of two of the MCMC processes according to the acceptance probability

$$\alpha = \left(1, \frac{\pi(\theta_k)^{\beta_j} \cdot \pi(\theta_j)^{\beta_k}}{\pi(\theta_j)^{\beta_j} \cdot \pi(\theta_k)^{\beta_j}}\right),$$

where i, j are arbitrarily chosen subscript and θ_x stands for the current states of the chain x having heat β_x. This scheme enables the exchange of samples between chains moving more freely on the parameter space and the original MCMC process sampling the density $\pi(\theta)$ while guaranteeing the correctness of this process.

This scheme can be interpreted by considering the exchange between two chains as being suggested by a proposal kernel characterised, or scaled, by its heat β_i. Therefore, the average acceptance probability of this kernel is directly tuned by β_i. Therefore, in most applications, parameters β are thus generally chosen as to guarantee exchange rates close to 50% between neighbour chains i and $i + 1$ (e.g. [4]). However, apart from these

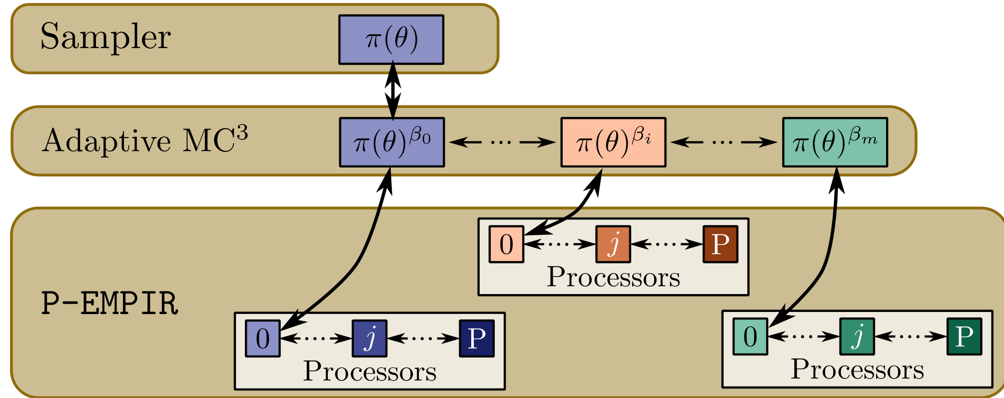


Figure 4.11.: Organization of the layer of parallelism in the final MCMC sampler. The first layer represents the sampling of $\pi(\theta)$. The second layer is the adaptive MC^3 that use m processors. The third layer is the m P-EMPIR algorithms having each P processors. The processor $j = 0$ of each P-EMPIR manages the heated chain of the MC^3 layer and the processor $j = 0$ of the heated chain $i = 0$ manages the sampling.

general rule of thumb, no guidance is defined for the choice of the values for β that must then be tuned by the user.

However, Miasojedow *et al.* [85] recently proposed an adaptive scheme for the MC^3 that enables the automated tuning of the β parameters such that the average acceptance probability of an exchange between two neighbouring chains i and $i-1$ targets a user defined probability. In HOGAN, this adaptive scheme is implemented using the stochastic approximation method already used for EMPIR (Eq. (4.5)). Therefore, as illustrated in figure 4.11, the current MCMC algorithm implemented in HOGAN is based on two layer of parallelism: one based on P-EMPIR and one based on this adaptive $MCMC^3$ method.

4.4. Assessing the performance of EMPIR and P-EMPIR implementation

In this section, the implementation of P-EMPIR is analysed and validated on several of the models described in chapter 1. In a first phase, the STD, MIXED and PCA variants of EMPIR are compared using multivariate normal based models as well as the PyRate model to highlight the contributions of each enhancement. In addition to these variants, two M-H algorithms using non-adaptive proposal kernels coupled with the pre-fetching method are used as reference:

- The PF method emulates user-defined proposal windows by using relevant but sub-optimal sizes of the proposals.
- The OPPF uses *optimal* proposal windows that are scaled based on Eq. (4.9) and

guidelines from [104] to target the optimal acceptance rate $\alpha_\star$.

In a second phase, the best variant PCA is challenged on several more complex and realistic tasks. The first is an analysis of a large plant fossil dataset [116] with the PyRate model. Then, the performance of P-EMPIR is compared with the state-of-the-art MrBayes[3] software [106] on codon models and on a phylogenetic reconstruction based on a nucleotide model. The complexity of these models and the fact that MrBayes already employs univariate adaptive proposal kernels offer an instructive benchmark. This benchmark ends with an overview of the performance gain brought by P-EMPIR on the phylogenetic reconstruction of a large dataset never analysed, until now, with Bayesian inference.

4.4.1. General experimental settings

Three types of measures were used to analyse the different properties of a MCMC sampler:

- The mixing of P-EMPIR was assessed by measuring the ACT and ESS. In an experiment, the average squared distance (ASD) between two samples was measured. This measure only represents the average travelled distance between iterations without considering the *randomness* of the samples.
- The rate of convergence of P-EMPIR on models having solely continuous parameters was measured by using the multivariate *potential* scale reduction factor $\hat{R}$ [19].
- The convergence of P-EMPIR on a phylogenetic tree distribution was diagnosed using the average standard deviation of split frequencies (ASDSF) as described in [76]. The split frequency represents the posterior probability of a taxa bipartition. The ASDSF estimates thus the stability of the split frequencies through iterations.

Using these measures, the overall performance of P-EMPIR was then determined using the speedup S defined as

$$S = \frac{\overline{g}(P, \mathtt{M})}{\overline{t}(P, \mathtt{M})} \div \frac{\overline{g}(1, \mathtt{M}_R)}{\overline{t}(1, \mathtt{M}_R)} \tag{4.12}$$

with the $\overline{g}(P, \mathtt{M})$ being the averaged measures (i.e. ESS or ASD), $\overline{t}(P, \mathtt{M})$ the averaged run time for P processors for the method M (e.g. PCA). In most of the cases, the measures of each method M was compared to the reference method $\mathtt{M}_R$ being PF with 1 processor. Under this circumstance, the speedup indicates the gain provided by the increase in ESS or ASD as well as the overhead caused by EMPIR and P-EMPIR. This overhead thus includes the cost of the learning phase of EMPIR and the communications between the processors plus the potential load balancing issues induced by P-EMPIR.

[3] All measures were made using MrBayes 3.2.5 compiled with SSE support and MPI enabled.

4.4.2. Validation on multivariate normal based models

Multivariate normal based models were used to validate the theoretical properties of the three variants STD, MIXED and PCA of EMPIR. Data were simulated using covariance matrices Σ representing uncorrelated and then correlated parameters such as to validate the variant of the properties on both types of parameter space. Under these settings the *optimal* proposal OPPF could be readily defined using the matrices Σ. Finally, given that the likelihood evaluations of this family of models can be obtained at nearly no computational cost and thus does not represent the target applications, the overhead of EMPIR and P-EMPIR were not taken into account so that only the raw ESS and ACT were analysed.

Validation against an optimal proposal.

The three variants were more effective than the sub-optimal proposal distributions (PF) at sampling the parameters $\theta = (\mu, V)$ of a simple model formed by independent normal distributions (Fig. 4.12a). Moreover, they were able to accurately learn the optimal proposals since their average ESS was comparable to the one of OPPF. The MIXED and PCA methods showed a similar behaviour than STD on this model since nearly no correlation were detected to trigger the switch from the adaptive proposal based on independent parameters (Eq. (4.11)) to the one exploiting correlations (Eqs. (4.7) or (4.8)).

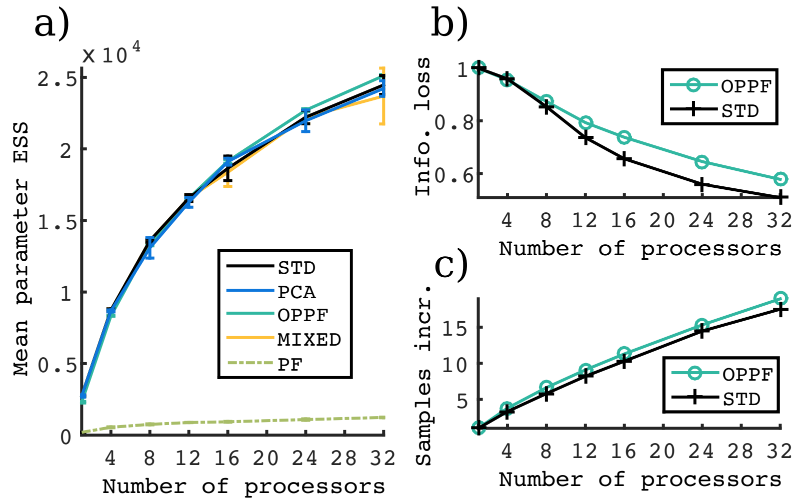


Figure 4.12.: Averaged results for independent normal models with $d = 20$, 3×10^3 iterations and 5 runs. Left figure shows the mean ESS per parameter in function of the number of processor for each method (error bars represent standard deviation). Right figure represents the decrease in sampling effectiveness (*top*) and increase of the number of samples per iteration (*bottom*) as the number of processors increases.

The improvements in ESS as a function of the number of processors was far from

linear and represents well the limitation defined in the efficiency equation (4.9) of the pre-fetching method. This equation balances the sampling efficiency $E^1(\alpha)$ and the estimated amount of samples produced per iteration $D(\alpha, P)$. The former is optimal for a set α while the latter depends on α but also on the number of processors P. Indeed, as the number of processors P grows, the amount of samples $D(\alpha, P)$ can be increased by improving the quality of prediction. This is achieved by reducing α which contradicts the previous statement that α should be constant to optimize $E^1(\alpha)$. This phenomena is well explained by the slow decay of the optimal target $\alpha_\star$ as a function of P, which leads to a rapid loss of sampling information per iteration (inverse of ACT; Fig. 4.12b) and a steady increase of samples kept per iteration (Fig. 4.12c).

Exploiting parameter correlations.

The PCA method outperformed significantly all other methods when sampling the parameters $\theta = (\mu)$ of multivariate normal distributions with several correlations expressed between parameters (Fig. 4.13a). All the adaptive methods showed comparable or superior average ESS than the OPPF method that performed independent normally distributed moves with the optimal variances. Since the PF method performed extremely poorly compared to the other methods, its results were discarded.

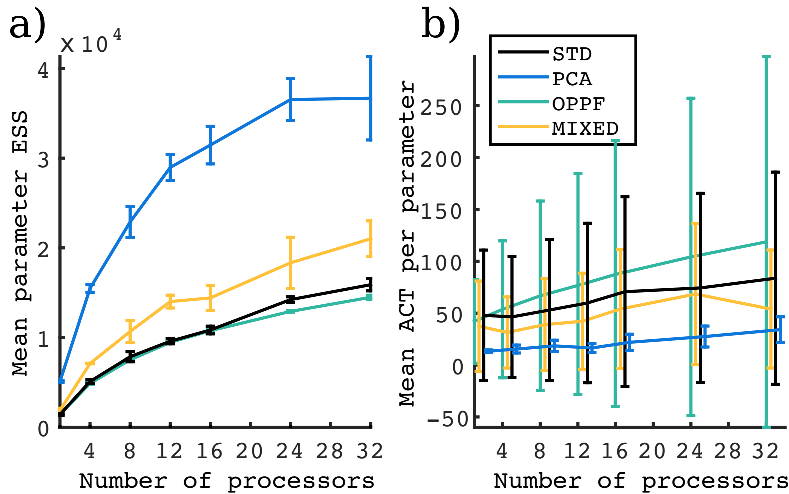


Figure 4.13.: Averaged results for a correlated multivariate normal with $d = 20$, 6×10^3 iterations and 5 runs. The figure on the left shows the mean ESS per parameter as a function of the number of processors for each method. The figure on the right represents the average ACT per parameter. Error bars represent the standard deviation.

While the MIXED method obtained slightly better results than OPPF, it underperformed compared to PCA due to the component-wise scaling of the covariance matrix Σ along each dimension, which impaired the information about correlations. This behaviour and

the one of the different methods is well illustrated by the standard deviation of ACT over the number of processor (Fig. 4.13b). By correctly exploiting the observed correlations, PCA was the only method to carry out a balanced sampling over all parameters. The other methods over and under-sampled some of them, which lead to high variance in the ACT.

4.4.3. Performance gain on PyRate model

The PyRate model was used to assess the performance of P-EMPIR in a controlled context. Indeed, the computational complexity of this model is growing as $\mathcal{O}(d)$ with d being the number of parameters and thus enables realistic experiments on large datasets to be achieved in reasonable computational time. The sampling performance on this model was not directly compared to the original python implementation given that the likelihood evaluations were measured to be more than 30 times faster in HOGAN.

Datasets

Two datasets were used to examine the performance of P-EMPIR on this model:

- The first is a dataset simulated using the simulator of the PyRate model [115]. Close to 4,000 fossil occurrences, assigned to 203 species, were simulated over a 30 million years (myr) time span. The preservation rate was set to 2.0 while two different speciation and extinction rates were defined over time. The speciation rate started at 0.4 from 30 to 20 myr and then reduced to 0.1, while the extinction rate started at 0.05 from 30 to 15 myr and then augmented to 0.4.
- The second is a dataset of plant fossils [116]. This dataset contains 22,415 fossil occurrences assigned to 443 plant genera. It spans over a hundred millions of years divided in 31 predefined epochs, which were defined by the stratigraphic geological time scale.

The first dataset was used to validate the previous observations made on the multivariate normal based models and to study the impact of P-EMPIR on the convergence of the MCMC process. The second dataset was used to challenge P-EMPIR on a realistic and computationally intensive problem.

Mixing efficiency

Due to the targeted learning, adaptive methods were more efficient at sampling the PyRate model with simulated data when compared to PF and even surpassed slightly OPPF (Fig. 4.14). Among adaptive methods, PCA showed the best performance because of the correlation that existed in the model. Indeed, while this amount of correlation is rather small, PCA was able to detect and exploit it.

Since the efficiency of the framework not only depends on the performance gain of P-EMPIR but also on its inherent overheads, the measured performance of P-EMPIR variants were compared with the reference non-adaptive method OPPF without pre-fetching

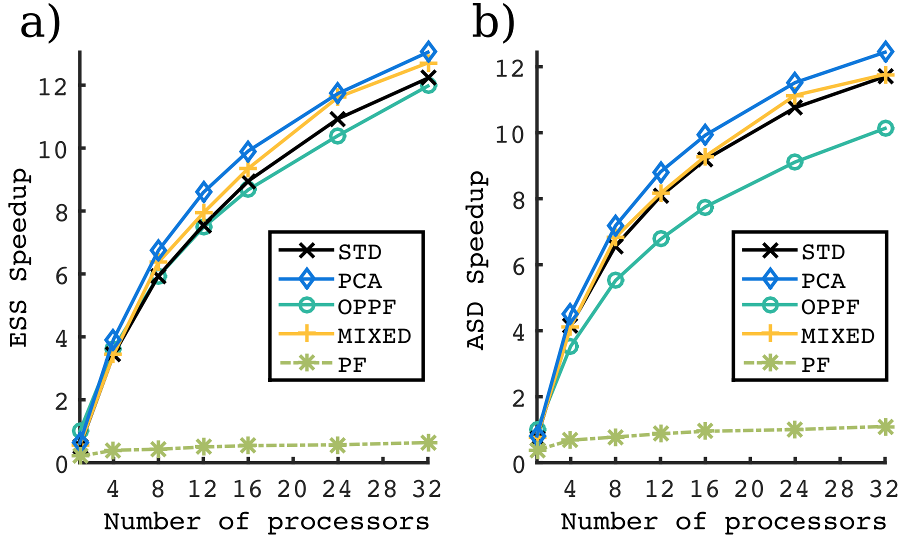


Figure 4.14.: Speedup for `PyRate` model on simulated data using `OPPF` with 1 processor as the reference. The figure shows the ESS, respectively ASD, speedup in function of the number of processors for each method. Settings for this simulation were $d \approx 400$, 4×10^6 iterations and 4 runs.

(see Eq. (4.12)). In this case, `OPPF` was determined using the average observed variance of each type of parameters (rates, times). In contrast, `PF` represented a sub-optimal choice with mean observed variance wrongly estimated by a tenfold factor.

The overheads due to processor communication and unbalanced loads of `P-EMPIR` amounted, on average, to 3% (4 processors) and 10% (32 processors) of the time spent per iteration. In contrast, the cost of the adaptive phase of `EMPIR` decreased from 13% (1 processor) to 6% (32 processors) and this decrease came from the faster convergence of the chain, the parallel processing of the component-wise scaling factors update and the added information provided by the adaptive process. This decrease of the adaptive phase cost directly highlights how the adaptive proposals benefit from being coupled with the pre-fetching method.

Rate of convergence

The multivariate potential scale reduction $\hat{R}$ measures highlighted that adaptive methods were much faster than their non-adaptive counterpart to reach the MCMC convergence on the `PyRate` model with simulated data (Fig. 4.15a)). This reduction of the burn-in phase provided by the adaptive methods is of utmost importance given that it limits the computational effort wasted at sampling irrelevant parts of the sampling space. Moreover, empirical observations showed that the convergence of the adaptive process occurred after the equilibrium was reached. This enables the use of the convergence of the adaptive process as an indication of the burn-in end and thus the start of the

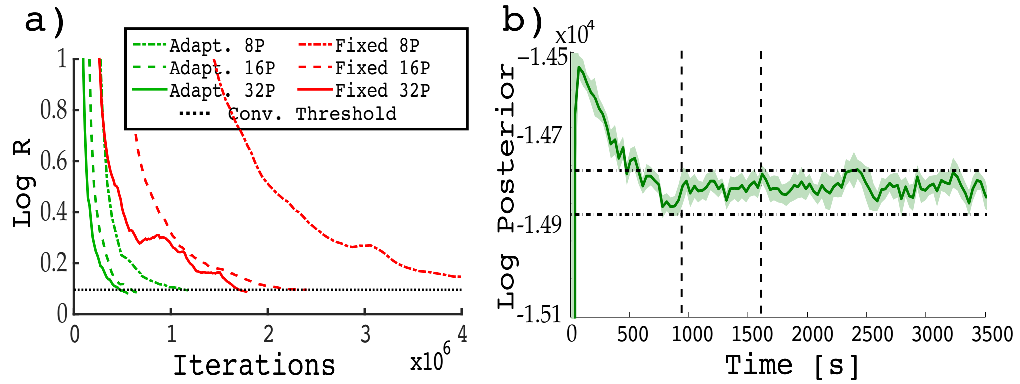


Figure 4.15.: The left figure shows the convergence of adaptive versus non-adaptive methods on the `PyRate` model by representing the *potential* scale reduction factor $\hat{R}$ based on 5 MCMC runs on simulated data. The black dotted lines represent the convergence threshold $\hat{R} = 1.1$. The right figure illustrates the evolution of `P-EMPIR`. The vertical dashed lines represents the convergence detection of the adaptive process.

sampling phase.

Indeed, as illustrated in figure 4.15b, a clear pattern about the learning process emerged from the large amount of experiments that were performed. During a first phase, the learned variables (Σ, λ, Λ) varied quickly during the burn-in, which created *wide* moves and improved the speed of convergence of the Markov chain. The end of this phase coincided with the first convergence detection of the learning process where parameters are assumed to be independent. After having chosen the appropriate proposal kernel, the learning phase restarted. Once the equilibrium was reached, the covariance matrix Σ converged to its final values while the scaling factors λ and Λ continued to be adapted. After some time, the second convergence of the learning process happened when λ and Λ reached values that maintained the target acceptance rate $\alpha_\star$ and thus the learning process ended.

Block size

The size of parameter blocks were previously identified as playing an important role in the efficiency of `EMPIR`. These theoretical postulates were confirmed by observations made during experiments on the `PyRate` model. On this model the use of blocks of parameters had a significant influence on the sampling performances. The ESS per second as a function of the size of blocks indicated that a block should have at least 2 parameters (Fig. 4.16). Indeed, blocks were made such as to group the two parameters related to the birth and death times of a species. This grouping enabled the strong correlation between both parameters to be exploited by `EMPIR`.

A second, less striking, observation could be made from these measures for blocks

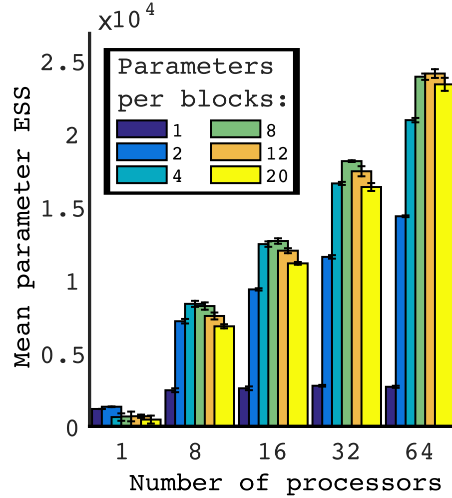


Figure 4.16.: Illustrations of the effect of block size on the PyRate model on simulated data. This figure shows the averaged effective sample size per second (with its standard deviation) of the PCA method in function of the number of parameters per block as well as the number of processors.

having at least two parameters: the optimal blocks size slightly changed with the number of processors employed (Fig. 4.16). For a low amount of processors, small blocks were more efficient given that the computational overhead of the adaptive phase was low. However, as the number of processors increased, larger blocks became more and more efficient due to the parallel estimation of the component-wise scaling factors that scale with the number of processors as well as the increased stability of the global scaling factor (Eq. (4.10)).

A real application.

P-EMPIR was then challenged on the analysis of a large dataset of plant fossils [116] with the PyRate model. The complex and heterogeneous parameter space of the model coupled with this empirical dataset highlighted the full potential of P-EMPIR. Adaptive methods revealed speedups between 2 to over 35 times when compared to the non-adaptive OPPF method depending on the number of processors used (Fig. 4.17a). Once again, PCA surpassed the other adaptive methods by exploiting the small amount of correlation that existed in the model.

In this application, the number of free parameters to estimate was approximately $1,000$, which makes it almost impossible for a user to guess a sensible set of proposal kernels for this real dataset. Indeed, the variances of the observed Markov chain for parameters identifying the species birth and death times ranged from 1 to 425 with an average value of 138 and standard deviation of 110. Using the average variance to define proposal kernels would result in some parameters being nearly never sampled. Actually moves configured with the observed average variance would be rejected almost all the

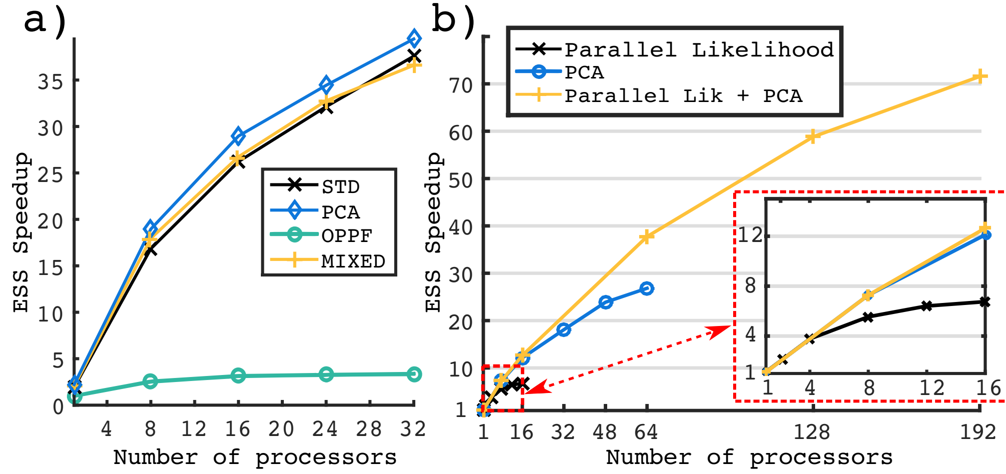


Figure 4.17.: Result for `PyRate` model on plant fossils data. The figure on the left shows the speedup of each method using `OPPF` with 1 processor as the reference. The right figure shows the parallel scalability of the framework when compared or combined to a parallel computation of the `PyRate` likelihood. Settings for these simulations were $d \approx 1000$, 8×10^6 iterations and 3 runs.

time when applied to parameters with variance one hundredfold smaller. An arbitrary size of ≈ 5 was therefore fixed for the proposal kernels of `OPPF`, which allowed each parameter to be sampled.

To illustrate the gains of `P-EMPIR`, the runtime required to sample this model with an average of 500 ESS per parameter are estimated under various pairs of methods and resources. Indeed, the runtime of such a task can be estimated by adding the time T_{BI} spent in the burn-in to the estimated sampling time T_S. The former is directly measurable, while the latter can be estimated based on the desired number N_{ESS} of ESS and the average observed time T_{ESS} required to produce one ESS. Therefore, the time to produce an average of 500 ESS using the `OPPF` method with one processor would be given by

$$\begin{aligned} T_{OPPF} &= T_{BI} + T_S = T_{BI} + N_{ESS} \cdot T_{ESS} \\ &= 1600 + 500 \cdot 47 = 2.35 \cdot 10^4 \text{ seconds} \end{aligned}$$

(i.e. seven hours). Switching to the `PCA` method, it would take only two and a half hours to reach the same level of ESS on one processor ($T_{BI} = 2670, S = 12$) and the time would reduce to 17 minutes if 16 processors ($T_{BI} = 230, S = 1.6$) would be used.

Beside the differences in computational time, the `OPPF` method would further produce a highly uneven sampling of all parameters due to the suboptimal proposal kernels. This unbalanced sampling was illustrated by the variation of ACT over all parameters observed during the experiments. An average ACT of 740 with standard deviation of

877 was obtained for the OPPF method, while the PCA method produced in contrast a far more reliable quality of sampling that was explained by its ACT variation of only 193 ± 273 for 1 processor and of 219 ± 177 for 16 processors.

Comparison with parallel likelihood.

The raw parallel performance gain of P-EMPIR was measured when confronted or combined to the parallel DAG computation available in HOGAN of the PyRate likelihood (see chapter 2). The performance of the PCA variant of P-EMPIR was compared when combined with *a)* a parallel likelihood evaluation, *b)* pre-fetching and *c)* both methods.

The variant *b*, identifying P-EMPIR, clearly surpassed the parallel likelihood one. It showed a nearly linear speedup up to 8 processors and notable gains up to 48 processors, while the parallel likelihood (variant *a*) showed a nearly linear speedup up to 4-6 processors before reaching a plateau with no gain in performance (Fig. 4.17b). The combination of both methods (variant *c*) showed the full potential of HOGAN by coupling several level of parallelism and thus by greatly increasing the parallel scalability and reaching speedups of ≈ 40 for 64 processors and ≈ 60 for 128 processors.

For these measures, the PCA proposal was used as the reference method in order to measure the parallel scalability without being biased by the gain induced by the adaptive proposals. Thus the speedup was measured according to equation (4.12) with $M_R = \texttt{PCA}$ using the PyRate model on the plant fossil dataset.

4.4.4. Performance gain on codon-substitution models

To illustrate the mixing performance of P-EMPIR on methods for molecular evolution [137], its performance was compared to the one of MrBayes on the model of codon substitutions *M2a*, which was developed to identify positive selection on protein coding genes. The computational cost of this model exceeds significantly the cost of the PyRate model since it requires matrix-matrix operations that depend on the alignment length and the phylogenetic tree size. It thus offers a completely different sampling challenge than the one of PyRate.

Particular care was taken to ensure an informative comparison of our implementation with MrBayes. The simulated phylogenetic tree was used as a fixed tree topology and the MC^3 method was disabled in both implementations. Under this circumstance, our implementations and MrBayes sampled the same set of parameters and both used adaptive proposals. Furthermore, the EMPIR variant closest to the univariate proposal kernel used in MrBayes was identified as the STD with only one parameter per block.

When comparing both methods under as identical setting as possible, HOGAN likelihood evaluations outperformed significantly the one of MrBayes on this model (see Apx. C.1). Therefore, a new hypothetical reference method M_R (Eq. (4.12)), identified as Hypothetical MrBayes, was defined by using the average reference ESS, $\overline{g}(1, M_R)$, as the one of MrBayes and the average runtime, $\overline{t}(1, M_R)$, as the one of the STD implementation with one parameter per block. The reference thus represent the effective sample size of MrBayes normalized by the runtime of our method with settings closest

to MrBayes.

Datasets

Two simulated datasets created using INDELible [36] were used to challenge P-EMPIR on this model. These datasets represented alignments formed of 100 codons simulated under mild purifying selection ($\omega = 0.8$) on phylogenetic trees having 16 taxa (Dataset 1) and 32 taxa (Dataset 2).

Mixing on codon-substitution models

First, using the measured ESS, the sequential executions of STD with one parameter per block were compared with the adaptive MCMC based algorithm implemented in MrBayes. Under this setting, the average ESS obtained were similar, yet slightly better for the STD variant (Fig. 4.18a, 4.18c). However, the PCA variant with 12 parameters per block showed a significant improvement in sampling performances. Indeed, the increase of average ESS when compared to MrBayes went from more than a twofold factor on a sequential execution up to a twentyfold factor with 32 processors.

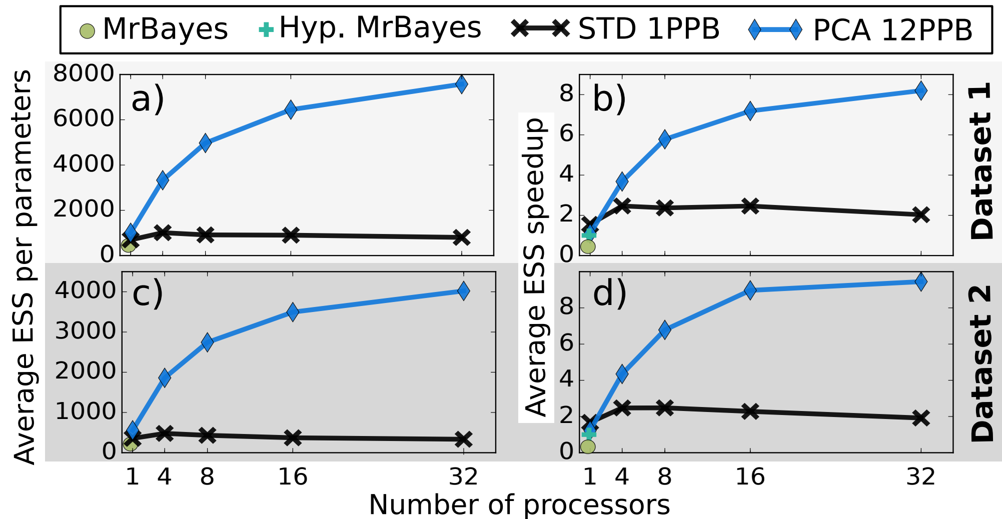


Figure 4.18.: Result for the ESS and ESS speedup comparison with MrBayes on the *M2a* codon substitution model with simulated data. Figures a) and c) show the average ESS per parameter on the first, respectively second, datasets for MrBayes, the STD method using single parameter moves and the PCA method with multiple parameters per block. Figures b) and d) illustrate the ESS speedup for the same methods on the same datasets using an hypothetical MrBayes implementation as reference. Simulations were run for $2 \cdot 10^5$ iterations.

While accurately representing the mixing efficiency of these methods, the ESS does not take into account the added computational complexity brought by moving multiple parameters at once. Indeed, HOGAN benefits from PLUs when few parameters are moved at once and MrBayes is taking advantage of a similar approach on a model-specific basis[4]. Therefore, given that PCA is moving more than one parameter per iteration, using the ESS speedup defined by Eq. 4.12 is more appropriate given that it encompasses these increases in computational cost by normalizing the ESS by the execution time.

The obtained ESS speedups (Fig. 4.18b and 4.18d) showed that P-EMPIR was still outperforming the hypothetical reference method from close to a twofold factor on a sequential execution and up to approximately a tenfold factor with 32 processors. However, in the sequential case, the simple STD variant was more efficient than the PCA variant. This is explained by considering the overhead of more costly partial likelihood computations and the added cost of learning parameter correlations. However, as soon as several processors were used, and thus the full potential of P-EMPIR was employed, the PCA variant was more efficient by proposing bolder moves and using the parallel resources to learn correlations more accurately and quickly.

4.4.5. Convergence of phylogeny inference

As a final illustration, P-EMPIR was challenged on the estimation of the posterior distributions of phylogenetic trees with their branch lengths and parameters of a general time reversible (GTR) model of nucleotide substitution [124]. This substitution model presents a lesser computational challenge than the codon model M2a. However, the major difficulty resides in properly sampling the space of potential phylogenetic trees. For that matter, two widely used tree proposals were implemented in HOGAN: the Stochastic Nearest Neighbor Interchange (stNNI) and the Extended Subtree Pruning and Regrafting (eSPR) in our framework.

P-EMPIR performance was compared to the one of MrBayes under two different settings. In the first setting, later referred as Ref MrBayes, only the stNNI and eSPR tree proposals were enabled to mimic our own implementation. In the second setting, later referred as Full MrBayes, the default configuration of MrBayes was used. This configuration employs several more evolved tree moves that further enhance the sampling of tree distributions.

Finally, all these different settings, including the ones of P-EMPIR were compared with and without the MC^3. The adaptive MC^3 method was disabled in P-EMPIR and a similar setting was used for the temperature β_i of the different auxiliary distributions $\pi(\theta)^{\beta_i}$.

Datasets

Two empirical DNA datasets used in [76] and available in TreeBASE[5] were used for this experiment:

[4]In MrBayes partial likelihood updates are hard-coded for update induced by change of a single branch-lengths of a phylogenetic tree.
[5]http://www.treebase.org

- The first one, M2017, with TreeBASE legacy ID M336, has 27 taxa and 1949 sites;
- The second one, M2152, with TreeBase legacy ID M520, has 67 taxa and 1098 sites.

These two phylogenies were selected to present a challenge in term of sampling while presenting an affordable computational cost.

A third dataset was used to give an insight of P-EMPIR's potential on a larger dataset that represents a challenge for state-of-the-art software dedicated to Bayesian inference for phylogenetic tree. This DNA empirical dataset published by Pyron and Wiens [99] encompasses over 2800 species of amphibians with more than 10000 sites per aligned sequences. While phylogenetic trees have already been inferred using ML methods for this dataset, none have been obtained using Bayesian inference until today.

Convergence on phylogenetic tree distributions

The PCA variant of P-EMPIR with 32 processors was able (Fig. 4.19) to converge towards the posterior distribution of the phylogenetic tree up to 10-20 times faster than its equivalent Ref MrBayes. When compared to the default settings of MrBayes (Full MrBayes) with more advanced tree proposals, P-EMPIR still showed faster convergence rate as soon as 4 to 8 processors were used. Against this MrBayes setting, the convergence speedup reached approximately a fourfold factor with 32 processors.

Figures 4.19a and 4.19c illustrate the speedup in ASDSF convergence on datasets M2017 and M2152, respectively, using Ref MrBayes setting without MC^3 as reference. Indeed, the Ref MrBayes setting showed similar execution time and convergence performance as our STD variant with updates on one parameter per iteration (see Apx. C.2). PCA slightly outperformed Ref MrBayes in the sequential case but was still far from the performance offered by Full MrBayes. However as the number of processors used increased, the performance of PCA exceeded both MrBayes settings.

To highlight the versatility of HOGAN, the same experiment was conducted with MC^3 enabled on both sofware. Figures 4.19b and 4.19d show the speedup in ASDSF convergence on both datasets with four parallel tempered chains. The trend shown in this second experiment is consistent with the previous results. The speedup of PCA increased with the number of processors used and surpassed the performance of Ref MrBayes as well as Full MrBayes.

The improvement on convergence rates brought by P-EMPIR on this problem is linked with the difficulty of moving through the phylogenetic tree space. Tree proposals are usually suffering from a low acceptance rate and are thus frequently rejected. Indeed, in these experiments the acceptance rates for tree proposals where lower than 0.05. While such moves are not currently adaptive, they remain easily predictable and therefore exploitable by the pre-fetching method. This assumption of low acceptance rate on tree proposals is true for most datasets, but it might falter on some of them. To get around this limitation, tree proposals could be made adaptive by tuning the extension probability (e.g. for eSPR) or the proposals on branch lengths that are jointly applied to tree modifications.

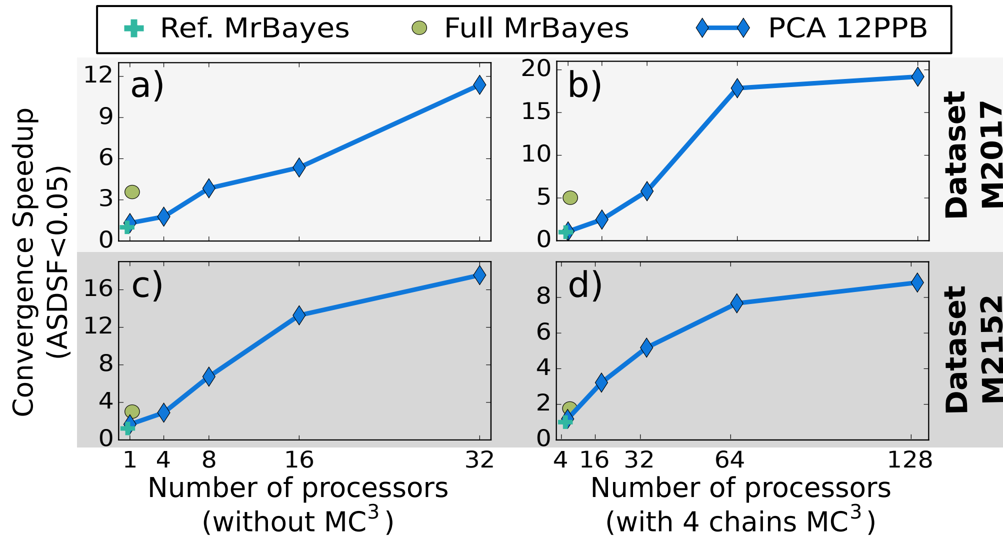


Figure 4.19.: Results for the ASDSF convergence speedup comparison with MrBayes when estimating the posterior distribution of the phylogenetic tree and the nucleotide substitution parameters on empirical datasets (TreeBASE M2017 and M2152). Figures a) and c) show the ASDSF speedup for the first, respectively second, datasets for two settings of MrBayes and our PCA method with multiples parameters per block. Figures b) and d) illustrate the ASDSF speedup on the same datasets when the previous methods are augmented with MC^3 .

Overview of the potential of P-EMPIR on a large phylogeny

These results suggest that P-EMPIR has a large potential on more challenging datasets. Indeed, the datasets used in previous experiments were easily analysable using existing software. The amphibian family dataset represents an actual challenge for both Full MrBayes and PCA and was thus used to give an insight of P-EMPIR potential. For this experiment, four separate runs of Full MrBayes using each four tempered chains were compared with four separate runs of PCA with 4 tempered chains having each 32 processors to their dedicated P-EMPIR.

Full MrBayes took more than five days to approach a likelihood plateau (Fig. 4.20), while PCA reached it in less than one day. This likelihood plateau did not actually represent the actual convergence of the MCMC process. Indeed, longer PCA runs showed that improvements at a smaller likelihood scale were still observable after several days. Moreover, it is worthwhile to mention that if PCA would be augmented with the advanced tree proposals present in Full MrBayes, further performance gains would be observed.

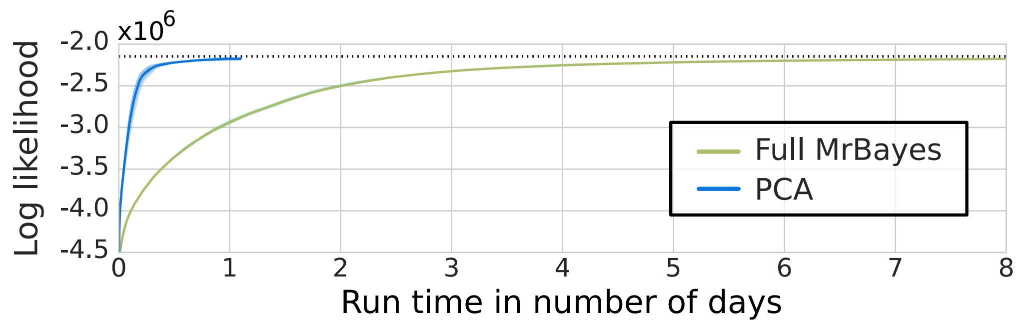


Figure 4.20.: Trace of the log likelihood when estimating the posterior distribution of the phylogenetic tree, branch lengths and nucleotide substitution parameters on an empirical dataset having 2800 species with 12 genes of the amphibian family. Average trace and 95% confidence interval are shown for 4 independent runs with 4 parallel tempered chains of `Full MrBayes` and `PCA` with 32 processors.

4.5. A step toward an automated block creation

The automated creation of good parameter blocks presents a difficult challenge that is usually delegated to the user or model developer. In this section, a promising mode-generic approach aiming to tackle this challenge is proposed by considering the creation of blocks as an optimization problem. This complex problem is first formalised and then simplified such as to produce an approachable minimization problem. Based on this definition of the problem, a heuristic method producing good solutions is proposed as a proof of concept. Finally, the integration of this method in `HOGAN` is detailed and then challenged on the inference of phylogenetic tree.

4.5.1. Formalization as an optimization problem

In section 4.3.1, three different factors were identified to play a key role in the definition of good parameter blocks. The first factor was identified as the size of the parameter blocks which directly defines the overhead of the `EMPIR` method as well as its stability and effectiveness. The second one was the importance of grouping parameters being highly correlated so as to reduce their dependency effects. The final one was related to the efficiency of the likelihoods evaluations, or PLUs, given that their computational cost highly depends on the parameters grouped together as discussed in chapter 3.

Therefore, the problem at hand consists in finding a good partitioning $\mathcal{D}$ of the set of parameters θ that minimises the multi-objective minimization problem defined as

$$\begin{aligned} \min_{\mathcal{D}} \quad & (f_1(\mathcal{D}), f_2(\mathcal{D}), f_3(\mathcal{D})) \\ \text{subject to} \quad & \mathcal{D} = \left\{ \mathcal{D}^{(i)} : i \in (1,..,m) \right\}, \\ & \emptyset \notin \mathcal{D}, \\ & \bigcup_{i\in(1,..,m)} \mathcal{D}^{(i)} = \theta, \\ & \bigcap_{i\in(1,..,m)} \mathcal{D}^{(i)} = \emptyset. \end{aligned} \tag{4.13}$$

The function f_1 defines the cost of PLUs, f_2 the overhead related to the size of blocks and f_3 a penalty of not grouping correlated parameters.

In this form this optimization problem is hardly approachable given that some of its functions are not entirely definable. Indeed, the function f_3 requires the availability of the exact or approximated full matrix covariance Σ. While the first is generally unknown, the second was previously defined as being overly expensive to approximate with the help of the adaptive phase of `EMPIR` and would be only accurate after the burn-in phase. A first simplification to the optimization problem is thus to omit the function f_3.

Difficulties arise also for the definition of the function f_2 and are mainly due to the overhead of the `EMPIR` method which is fluctuating during the adaptive phase dependently on the model analysed. Moreover, the exact gain in mixing coming from `EMPIR` is hardly assessable during a MCMC run because of the expensive computational cost of

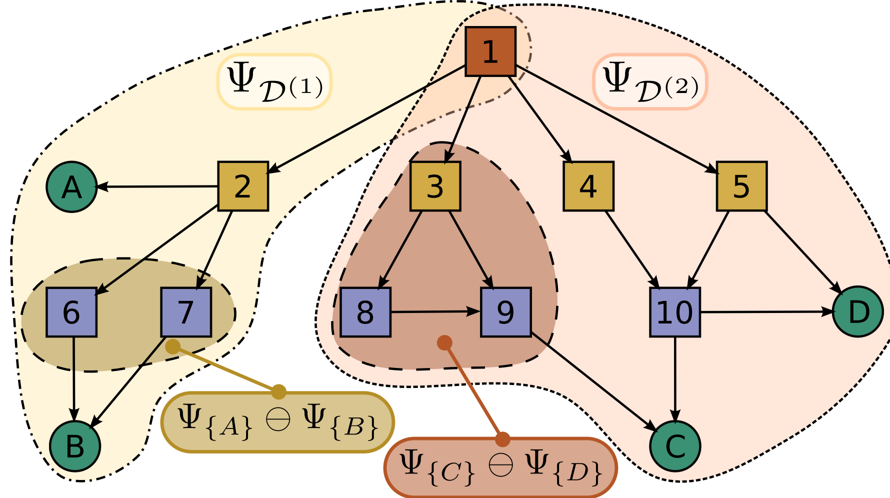


Figure 4.21.: Illustration of two partitions, $\mathcal{D}^{(1)} = \{A, B\}$ and $\mathcal{D}^{(2)} = \{C, D\}$ on a simple directed task graph. The symbol $\ominus$ defines the symmetric difference of two sets.

the ACT measure. However, contrary to the previous function this one is too important to omit. Therefore, f_2 is overly simplified such as to maintain the block size close to an empirically defined value L, such that

$$f_2(\theta, \mathcal{D}) = \sum_{\mathcal{D}^{(i)} \in \mathcal{D}} \left(L - |\mathcal{D}^{(i)}|\right)^2 .$$

Fortunately, the function f_1 is definable thanks to the DAG representation of likelihoods used in `HOGAN`. Indeed, the cost of PLUs can be estimated using the same scheme as in chapter 3. The proof of concept presented in this section for an automated and model-generic creation of parameter blocks is thus based on the minimization of the computational cost of PLUs.

Minimising the PLUs cost by adequately choosing parameter blocks

The importance of correctly grouping parameters together is clearly apparent when considering the partitioning $\mathcal{D}$ of the simple directed task graph illustrated in figure 4.21. The partitioning proposed in the figure results in two partitions: $\mathcal{D}^{(1)} = \{A, B\}$ and $\mathcal{D}^{(2)} = \{C, D\}$. Assuming a unit cost for each task, the computational cost of the first and second partition would present a computational cost of 4 and 7, respectively.

If the partitions chosen would have been $\mathcal{D}^{(1)} = \{A, C\}$ and $\mathcal{D}^{(2)} = \{B, D\}$, then the computational cost of PLUs would have been way more expensive. Indeed, the first partition would require 7 nodes to be recomputed, while the second one would require 10 nodes. Compared to the one presented in the figure, this partitioning would thus be on average (17/11) times costlier.

Therefore, minimising the overall PLUs cost is a good starting point for the creation of parameter block. Based on the definition of chapter 3 the cost of a PLU can be defined as

$$\mathcal{C}\left(\mathcal{D}^{(i)}\right) = \sum_{v_s \in \Psi_{\mathcal{D}^{(i)}}} \chi_s \tag{4.14}$$

with the cost χ_s of a task v_s either representing a unit cost (Eq. (2.13)) or the observed computational time (Eq. (2.13)). Using this cost $\mathcal{C}$, the function f_1 is simply defined as the sum of each partitions cost such that

$$f_1(\mathcal{D}) = \sum_{\mathcal{D}^{(i)} \in \mathcal{D}} \mathcal{C}\left(\mathcal{D}^{(i)}\right). \tag{4.15}$$

Based on the previous definition of f_1 and f_2, the simplified problem identifying the creation of blocks presents several similarity with the balanced graph partitioning problem [20]. This NP-Hard problem focus on the partitioning of vertices $\mathcal{V}$ of a graph guaranteeing that the partitions sizes fit within a prescribed threshold. Moreover, this partitioning must minimize an objective function defined on the edge weights $\{w_i : e_i \in \mathcal{E}\}$.

The simplified problem of creating blocks falls into this category. Indeed, if each parameter θ_i is considered as a as vertex of $\mathcal{V}$ then the edges weights can be defined as the *computational distance* $\delta(i,j)$ separating parameters θ_i and θ_j. This computational distance $\delta(i,j)$ represents the difference of the tasks to recompute during PLUs induced by change of parameters θ_i and θ_j, such that

$$\delta(i,j) = \sum_{v_s \in \Psi_{\mathcal{A}_{i \ominus j}}} \chi_s \quad \text{with} \quad \Psi_{\mathcal{A}_{i \ominus j}} = \Psi_{\{\theta_i\}} \ominus \Psi_{\{\theta_j\}}, \tag{4.16}$$

and $\ominus$ representing the symmetric difference between two sets.

These distances are illustrated in figure 4.21 for parameters grouped in each partitions. Parameters A and B have a distance of 2 tasks while parameters C and D have a distance of 3 tasks. However, the hypothetical partition grouping parameters B and C would have a distance of 9 tasks identified as

$$\begin{aligned}\Psi_{\{B\}} \ominus \Psi_{\{C\}} &= \{v_1, v_2, v_6, v_7\} \ominus \{v_1, v_3, v_4, v_5, v_8, v_9, v_{10}\}, \\ &= \{v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}.\end{aligned}$$

Using this definition of the distance, the minimization of the PLUs cost can be reinterpreted as the minimization of the *intra-partition* edges weights, or the cost of grouping parameters, such that

$$f_1(\mathcal{D}) = \sum_{\mathcal{D}^{(k)} \in \mathcal{D}} \left[\sum_{\theta_i \in \mathcal{D}^{(k)}} \left(\sum_{\substack{\theta_j \in \mathcal{D}^{(k)} \\ j > i}} \delta(i,j) \right) \right]. \tag{4.17}$$

Unfortunately, this objective function is different from the most-widely studied ones that aim to minimise the scheduling of tasks so as to minimise the communication between

processors [11]. This problem is equivalent to minimizing the *inter-partition* edges and therefore readily available implementation of a such method [66] cannot be used for the problem at hand.

4.5.2. A hierarchical clustering based heuristic

A custom heuristic based on graph clustering methods [109, 70] was designed to produce good solutions to the simplified problem. The concept of this heuristic is to first apply a graph clustering algorithm using the edge weight matrix containing all the parameter pairwise distances $\delta(\cdot, \cdot)$. Then in a second step, the resulting clusters are reorganised by either splitting or merging them such as to obtain an adequate partitioning of the parameters.

The choice of the family of clustering algorithms employed was defined so as to simplify the second step. Indeed, by using a hierarchical clustering method, the second phase can heavily rely on the cluster structure expressed in the resulting dendrogram. The `UPGMA` algorithm [118] was thus implemented for the graph clustering. Starting from clusters $\mathcal{B}_k$ composed of a single parameter θ_k, this method iteratively agglomerates the two clusters of parameters, $\mathcal{B}'$ and $\mathcal{B}''$, being the closest with respect to their averaged *inter-cluster* node distance defined as

$$\frac{1}{|\mathcal{B}'| \cdot |\mathcal{B}''|} \sum_{\theta_i \in \mathcal{B}'} \sum_{\theta_j \in \mathcal{B}''} \delta(i, j). \tag{4.18}$$

This agglomeration criterion clearly fits the problem defined in equation (4.17).

Once the dendrogram τ, identifying each parameter with a leaf node, is obtained using the `UPGMA` method, algorithm 9 is applied so as to define the partitioning $\mathcal{D}$ of the parameters. This algorithm searches in τ for the subtree having the number of leaf nodes L_s the closest to L and forms a partition $\mathcal{D}^{(i)}$ with them. This subtree is then pruned and this step is repeated until either no more leaf nodes remain in τ, or the number L_s is too small compared to L in which case the remaining leaf nodes are distributed among all the partitions.

An illustrative clustering and partitioning resulting from the application of this heuristic is shown in figure 4.22. This heuristic provides good parameter blocks in regard of their PLUs cost at a complexity of $\mathcal{O}(d^3)$ with d being the number of parameters. This cost is affordable on models having computationally expensive likelihoods or few number of parameters. However, for very large number of parameters, this approach would require a significant computational effort.

An experiment on the problem of phylogeny reconstruction

This heuristic method was compared with two others strategies on the task of defining parameter blocks of size L for phylogenetic inference models analysing the large amphibian dataset [99]. The first strategy represents a blind choice of the parameter blocks and is thus simulated by randomly formed blocks of the desired size. The second strategy

Algorithm 9 Definition of the partitioning $\mathcal{D}$ with block of size $L \pm (\epsilon_{max} \cdot L)$ from the dendrogram defined as $\tau = \{\mathcal{V}, \mathcal{E}\}$.

```
i = 1
while |V| > 0 do
  // Returns the subtree rooted at v_s, and its number of
  // leaves L_s, with the number of leaves the closest to L
  {v_s, L_s} = findClosestSubtree(τ, L)
  ε = |L − L_s| / L
  if ε < ε_max then
    Form D^(i) with leaves of subtree rooted at v_s
    Prune subtree rooted at v_s from τ
    i = i + 1
  else
    if L − L_s < 0 then
      Distribute remaining leaves equally to D^(·) ∈ D
      V = ∅
    else
      Relax tolerance: ε_max = ε
    end if
  end if
end while
```

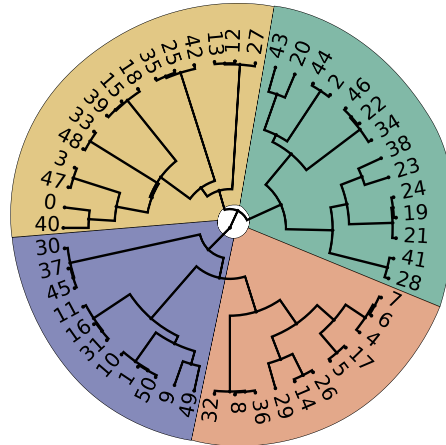


Figure 4.22.: Illustration of the hierarchical clustering and then partitioning ($L = 12$) of the parameters on a model for phylogenetic inference analysing dataset M2017. Each color represent a different parameters partition.

is the one that was used during the previous experiments on P-EMPIR. Indeed, a model-specific block creation strategy, based on the observations of chapter 3, was used for the phylogenetic tree based model. In order to minimise the cost of PLUs, branch length parameters were ordered according to a post-order depth-first search traversal and then partitioned in equally sized subsets, while parameters of the evolutionary model were grouped in a single block.

However, in addition of being model-specific, this approach was identified during the current experiment as being problematic for phylogenetic inference models. Indeed, parameter blocks were initially defined in function of the arbitrarily chosen starting phylogenetic tree structure. After several iterations and thus several accepted tree moves, the tree structure was significantly altered and so was the related structure of the DAG representing the likelihood evaluations. Therefore the, once adequate, parameter blocks defined in function of the initial DAG structure were then hindering the performance of the likelihood evaluations.

This phenomena is illustrated in figure 4.23 by showing the average likelihood evaluation time during a MCMC run with the three strategies. The one creating random blocks was shown as constantly having a poor performance compared to the others. The model-specific strategy started with good parameter blocks that however quickly deteriorated as the phylogenetic tree structure evolved. The model-generic heuristic was periodically applied during the MCMC run such as to re-optimise the blocks as the tree evolved. This strategy showed the best results on the long run and furthermore the quality of automatically created blocks were close to the initial quality of the ones obtained with the model-specific approach.

This experiment highlights the challenge of defining good blocks in general, and even more so for models having an evolving DAG structure. Furthermore, it gives an insight

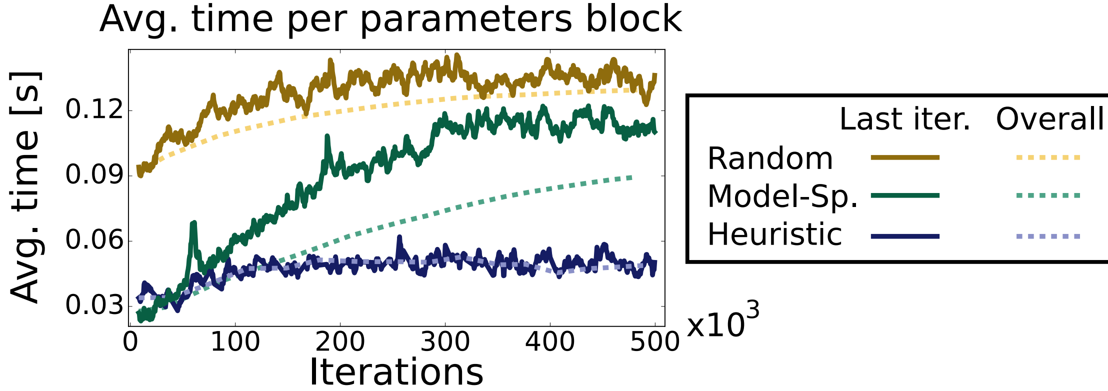


Figure 4.23.: Illustration of the performance of different block creation strategies on the phylogenetic inference of the large amphibian dataset. These measures are representing the averaged likelihood evaluations time observed following the update of each of the parameter blocks. The full line defines this average time by only taking account of the last 20 likelihood evaluation times of each block, while the dashed line encompasses the whole history of likelihood evaluation times.

on the quality of the partitioning $\mathcal{D}$ obtained for a complex model having close to 6000 parameters with the proposed heuristic. However, these results mainly serve as a proof of concept for this approach that would greatly benefit from a more thorough theoretical and practical analysis. For instance, redefining parameter blocks on the fly during MCMC runs poses a serious challenge with respect to the adaptive proposal kernels and their approximated covariance matrix Σ and scaling factors λ, Λ and is thus the subject of the next section.

4.5.3. Adaptive parameter blocks creation

Optimising the parameter blocks during a MCMC runs is a perilous task given that it may tamper with the adaptive proposal kernels used in `P-EMPIR`. However, the previous experiment identified that such an approach could significantly improve the sampling speed of complex statistical models and thus a strategy was designed to integrate the automated creation of blocks in the adaptive scheme. The required steps permitting this integration while guaranteeing the correctness and efficiency of `P-EMPIR` are now detailed.

Two issues arise whenever blocks are redefined. The first is concerned with the correctness of the adaptive process. Indeed, the stochastic approximation method used in equation (4.5) relies on a sequence of multipliers γ_k that decreases as the adaptive process progresses such as to guarantee its convergence. This property is of utmost importance given that it determines if the MCMC process is asymptotically converging to the target distribution $\pi(\theta)$.

Therefore, a special care must be taken during the creation of new blocks to guarantee this convergence. Given that each approximated variable Σ, λ or Λ of a parameter blocks has a dedicated stochastic approximation process with an internal counter k, the newly created blocks must insure that the new counters $\hat{k}$ are at least as big as the previous one. Assuming a newly created block $\hat{\mathcal{D}}^{(j)}$ based on the original ones $\mathcal{D}^{(\cdot)}$, then its counter $\hat{k}_j$ for a variable , e.g. λ, has to be defined as

$$\hat{k}_j = \max\left\{k_i : \theta_l \subseteq \mathcal{D}^{(i)} \wedge \theta_l \in \hat{\mathcal{D}}^{(j)}\right\}.$$

The second issue concerns the efficiency of the adaptive process since variables Σ, λ and Λ learned beforehand could be lost during the creation of new blocks. To minimise this loss, the variance Σ_{ii} and the scaling factors λ and Λ_i of each parameter θ_i are summarised and then employed to define the new blocks variables. Unfortunately, the off-diagonal elements of the covariance matrices Σ are lost in this process. Given that this information is critical whenever correlations between parameters are observed, parameter blocks having been detected as having correlations by `EMPIR`, and thus employing a multivariate proposal kernels, are preserved.

Algorithm 10 Simplified pseudo-code for the integration of the automated block creation to `P-EMPIR`. Signal the *convergence* of this process the last n_{MAX} attempts failed to improve $\mathcal{D}$.

`// Starting from partitioning` $\mathcal{D}$
Remove partitions $\mathcal{D}^{(i)}$ detected as correlated from $\mathcal{D}$
if Observed likelihood evaluation times are accurate **then**
  $\hat{\mathcal{D}} \leftarrow$ Apply heuristic on $\theta_i \in \mathcal{D}$ with observed time (Eq. (2.14))
else
  $\hat{\mathcal{D}} \leftarrow$ Apply heuristic on $\theta_i \in \mathcal{D}$ with unit node cost (Eq. (2.13))
end if
`// Check if the new partitioning` $\hat{\mathcal{D}}$ `improves the blocks`
if $\mathcal{C}(\hat{\mathcal{D}}) < \mathcal{C}(\mathcal{D})$ **then**
  Re-factor blocks according to $\hat{\mathcal{D}}$
  • Define stochastic approximation counters $\hat{k}$
  • Recycle variables $\Sigma_{ii}, \lambda, \Lambda_i$
  $n_{fail} = 0$
else
  $n_{fail} = +1$
end if
return $(n_{fail} = n_{MAX})$

Using the aforementioned recycling scheme of the old blocks, the adaptive creation of blocks (`ACB`) can be integrated in `HOGAN` as defined in the simplified pseudo-code of algorithm 10. In function of the number of blocks, this algorithm is periodically called during a MCMC run until the convergence of the parameter blocks has been signalled.

Experiments on the problem of phylogeny reconstruction (cont.)

The previous experiments on the task of defining parameter blocks of size L for phylogenetic inference models was continued in order to assess the performance of the `ACB`. However, conversely to the previous experiment, all three datasets previously defined, the datasets M2017 and M2152 (from TreeBASE) as well as the amphibian dataset, were employed for this experiment.

Firstly, the convergence of the adaptive proposal kernels designed in `P-EMPIR` and thus of the block creations was assessed on the two reasonably sized datasets M2017 and M2152. Multiple runs on both instances proved the convergence of the adaptive phase to be reached without problems. However, due to the reconstruction of parameter blocks and thus of the loss of information on the covariance matrix, the adaptive phase of `P-EMPIR` took on average 10% more iterations to reach convergence.

Secondly, the improvements coming from the `ACB` were estimated by measuring its speedup with respect to the random creation of parameter blocks. The improvements coming from the model-specific strategy employed for the initial experiments on the phylogeny reconstruction were also compared to the random strategy. The comparison of both performance gains coming from the `ACB` and the model-specific strategies gives thus an insight on the additional speedups that could be applied to `P-EMPIR`.

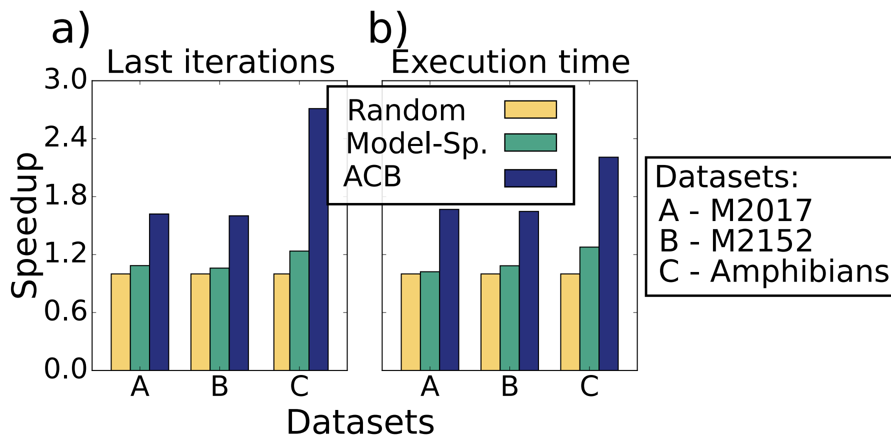


Figure 4.24.: Measures of the speedup with respect to the random strategy of the model specific and adaptive creation strategies on the phylogenetic inference model. The left figure shows the speedup of the averaged last 20 likelihood evaluation times. The right figure represents the speedup of the total execution time of the MCMC runs. Experiments on datasets A and B were conducted with 1 processor per MCMC run, while the ones on datasets C used 4 processors for `P-EMPIR`.

The experimental setting consisted on launching MCMC runs for fixed number of iterations that were long enough to observe the convergence of the adaptive phase on TreeBASE datasets. The number of iterations was empirically fixed to $5 \cdot 10^5$ for the

amphibian dataset. Based on these runs, two different speedups were considered. The first only took into account the averaged time over all blocks of the last 20 PLUs that were induced by the update of their parameters. The second measured the total execution time of the MCMC runs in order to also include the potential overhead incurring from the adaptive method.

As illustrated in figure 4.24a), the measured speedups over the last likelihood evaluations brought by the `ACB` strategy showed a significant improvement when compared to the random and model specific strategies. Indeed, as identified in figure 4.23, the performance of the model specific strategies declined on these long MCMC runs.

The speedup over the total execution time represented in figure 4.24b) corroborated these observations and highlighted that the overhead of the `ACB` method was small compared to the likelihood evaluations cost. Indeed, the time spent applying algorithm 10 was measured on the three datasets and only represented an additional cost of 0.5% of the total execution time.

Therefore, the small decrease of speedup noticeable for the amphibian dataset on figure 4.24b) can not be explained by the overhead of the `ACB` strategy. More thorough investigations are required to accurately identify the cause of this effect, however two potential causes are proposed. The first cause could be related with load balancing issues inherent to `P-EMPIR` given that this dataset was the only one using `P-EMPIR` with multiple processors. The second cause could be associated with the disparity of computational time between the PLUs induced by branch-length parameter blocks and the one induced by the evolutionary model parameter block. Indeed, for the two TreeBASE datasets these times diverged by a twofold factor, while for the amphibian dataset divergence of two orders of magnitude were measured.

In conclusion, the proof of concept proposed in this section led to performance gains that could significantly improve the sampling efficiency of `P-EMPIR` on a model-generic basis. For instance, the speedups of convergence observed in section 4.4 on the phylogenetic reconstruction model could be nearly doubled. However, before hastily drawing conclusion from these promising gains, more experiments should be conducted to fully assess the effect on the convergence and sampling efficiency of `P-EMPIR` when using the `ACB` scheme.

4.6. Summary

In this chapter, means of improving the efficiency of MCMC methods are studied. Two novel strategies based on existing concepts in computational statistics, adaptive proposals and pre-fetching, are proposed and their implementation is discussed. These methods are thoroughly tested on several models and datasets. Finally, a proof of concept for an automated parameter blocks creation strategy is presented.

Model-generic

The theoretical aspect of a novel M-H algorithm based on a enhanced adaptive proposal, `EMPIR`, and its synergistic coupling with the pre-fetching method, entitled `P-EMPIR`, are detailed:

- `EMPIR`: a family of adaptive proposals that combines local and global scaling factors to efficiently balance the sampling of continuous parameter blocks.
- `P-EMPIR`: a parallel M-H method composed of pre-fetching and `EMPIR` that are coupled by tuning the acceptance rate according to an existing performance model. This scheme results in an optimal setting for pre-fetching and an increase in efficiency for `EMPIR`.

The implementation of these methods is then discussed so as to provide an efficient and practical MCMC sampler by correctly exploiting the PLUs:

- `EMPIR` efficiency is improved by using parameter blocks, detecting the adaptive process convergence and automatically choosing the appropriate proposal kernel in function of the observed parameters correlations.
- `P-EMPIR` scaling and interaction with PLUs are made efficient by defining a reflection scheme for multivariate rotated proposals and by defining a strategy for the load balancing of PLUs.

The complex problem of automatically creating parameter blocks is then addressed:

- The creation of parameter blocks is formalised as a multi-objective optimization problem and simplified to produce a tractable problem based on the DAG representation of likelihoods.
- A greedy heuristic is proposed for the optimization of this constrained combinatorial problem.
- This strategy is made adaptive by integrating it in the adaptive phase of `P-EMPIR`.

The resulting adaptive parameter blocks creation scheme presents a crucial instrument for *evolving models* by using the DAG representation to bridge the structure of parameter blocks with the structure of the model. Evolving models encompass the models for phylogeny reconstruction that change with tree proposals and more importantly the models using transdimensional Markov chains [117] that enable model selection within MCMC methods.

Model-specific

Several evolutionary biology models were employed to study the behaviour and performance of EMPIR and P-EMPIR on synthetic and empiric datasets. An excerpt of the results are here synthesised:

- P-EMPIR accelerated upto 36 times with 32 processors the sampling of the posterior distribution of the PyRate model on a large empirical dataset when compared to a sequential non-adaptive proposal.
- On this same setting, the combination of parallel likelihood evaluations and P-EMPIR resulted in an unprecedented sixtyfold speedup with 128 processors.
- On codon-substitution site-models detecting positive selection, HOGAN was compared to the state-of-the-art software MrBayes. Likelihood evaluations with HOGAN were significantly faster (2x) on these models. On top of these gains, P-EMPIR increased the sampling efficiency by upto a tenfold factor with 32 processors.
- On a model for phylogenetic tree reconstruction, P-EMPIR accelerated the convergence toward the correct tree distribution by upto a twentyfold factor with 32 more computational resources than the reference software, MrBayes.
- A large empirical dataset counting approximately 3,000 species was used to illustrate the potential of P-EMPIR at the task of inferring phylogenetic tree distributions. P-EMPIR with 32 processors achieved in less than 20 hours the same level of progress than MrBayes in 8 days.
- The proof of concept of the adaptive parameter blocks creation revealed that the observed performance gains of previous experiments on the model for phylogeny inference could be nearly doubled.

5. Large scale analyses

Large scale analyses appear under various forms and can be motivated by the amount of datasets, the hypotheses challenged, or the statistical method employed. Indeed, both statistical methods implemented in `HOGAN` require experiments to be replicated so as to increase the confidence level in the results obtained. The number of replicas n_r varies as a function of the model analysed as well as the method employed: MCMC methods usually require rarely more than ten replicates to ensure the convergence of the process [14], while ML estimation may need up to several hundreds of replicas on complex models to properly infer confidence values [93].

The amount of replicas hardly justifies the *large* qualifier in *large scale analyses*, however its combination with the number of different hypotheses n_h used for model comparisons may deserve it. For instance, the branch-site model for the detection of positive selection always requires two different hypotheses to be compared: the null hypothesis that does not model positive selection and the alternative one that does.

For a thorough analysis with this model on a given dataset, all the possible combinations of branches of the phylogenetic tree would have to be independently tested for positive selection. In the more realistic case where solely the n_b branches of the tree would be tested separately, the number of independent analyses would amount to $n = (n_r \times n_h \times n_b)$.

However, this is not all: the ever growing amount of molecular data must be taken into account in order to provide the whole picture. Indeed, this richness in data has motivated the creation of database containing the summary of statistical analysis for several thousands of molecular datasets [122]. Such is the case of `Selectome` [98, 87], a database indexing the results of more than 50,000 independent analyses conducted with the branch-site model for the detection of positive selection.

These large scale analyses are thus characterised by hundreds to thousands of computational tasks that, if run sequentially, may represent a time-wise insurmountable computational challenge without even considering the manpower required by such a laborious project. Fortunately, the large amount of computational resources available nowadays provides the necessary means to tackle such an enterprise.

The dynamic load balancing algorithm implemented in `HOGAN` that enables to compute several independent computational tasks on supercomputers or large clusters while still benefiting from the previously presented parallel statistical methods is shortly described in this chapter. Following its description, an overview of the original application for which this algorithm was implemented, namely the creation of a database of coevolution, is presented.

5.1. Dynamic load balancing for HPC

The management of large scale analyses in HOGAN must cohabit with its parallel strategies that enhance statistical methods. These parallel strategies are meant to exploit heterogeneous computing resources having fast interconnection networks. Therefore, the solution chosen for the scheduling of large scale analyses has to enable the use of high-performance computing (HPC) architectures.

While the scheduling of several independent tasks could have been delegated to the job scheduler of the clusters, the choice was made to integrate it in HOGAN for two reasons: to enable the design of large analysis within the framework as well as to broaden the applicability to a wider range of supercomputers. Indeed, this approach enables an entire large analysis, which is composed, for example, of multiple hypothesis and replicates, to be defined in HOGAN. It thus removes the responsibility of the laborious management of several hundreds of jobs from the end-user. Furthermore, it also enables the use of supercomputers, that enforce a minimum amount of processors to be booked, for the computation of several small to mid-sized tasks.

The scheduling method to implement was then defined by the nature of the computational tasks processed in HOGAN. During a run, these tasks identify the statistical analysis of various datasets with different models. Therefore, their computational cost is heterogeneous and more importantly is unidentifiable beforehand since the convergence time of a statistical analysis with MCMC or ML methods is determined by the unknown shape of the parameter space. Under these conditions, the use of a dynamic load balancing algorithm is advocated to ensure that no processor is idle while others are overwhelmed by their computational load.

Finally, in prevision of the use of several hundreds of processors, the choice was made to implement an asynchronous and distributed load balancing algorithm. Indeed, a centralised algorithm, that delegates the scheduling to a single *master* processing unit, would lead to a bottleneck situation with the master being overwhelmed by large amounts of requests and a synchronous algorithm would have induced the processors having small computational tasks to be hindered by processors having more expensive ones.

5.1.1. A receiver initiated diffusion algorithm

Among the algorithms proposed by Willebeek-LeMair and Reeves [133], the *receiver initiated diffusion* (RID) algorithm was selected for its superior performance. The concept of this algorithm is rather straightforward: whenever a processor $p \in (1..P)$ detects that its load l_p is significantly inferior to the load l_q of its neighbour $q \in \psi(p)$, with $\psi(p)$ representing the set of its neighbours, then processor p asks processor q to share a part of its load proportional to the load imbalance $l_q - l_p$. For that strategy to work, processors have to communicate periodically their load to their neighbours.

Based on this simple concept this algorithm can be defined by two sets of operations encapsulating the processing of a task t_i. The operations happening on processor p after computing task t_i corresponds to algorithm 11 and begin by the update of the processor p load. Following this update, messages sent to processors p by its neighbours

Algorithm 11 Operations applied by processor p after having computed task t_i. The function $\mathcal{C}(t_i)$ defines the cost of task t_i and ϵ the absolute threshold requiring a processor to signal that his load has changed.

$l'_p = l_p$
Update load $l_p = l_p - \mathcal{C}(t_i)$
if Job requests answered **then**
  Update task queue, local load l_p and task request counter j_p
end if
Answer pending task requests
for all $q \in \psi(p)$ **do**
  if $l_p \ll l_q$ **then**
    Send job request to q for a load $l \propto l_q - l_q$
    Increment task request: $j_p = j_p + 1$
  end if
end for
if $|l_p - l'_p|/l'_p > \epsilon$ **then**
  Send load l_p to all $q \in \psi(p)$
end if

are processed. Then, this processor diffuses its load according to the potential pending task requests from its neighbours and, if significant load imbalances are detected in its neighbourhood, sends job requests to the relevant neighbours. Finally, the resulting load l_p of processor p is signalled to its neighbours whenever it has significantly changed.

The other operations happening on processor p before the computation of a task t_i are detailed in algorithm 12 and mostly deal with the lack of locally available tasks for computation. In this situation, the processor waits for its emitted task requests to be answered. If tasks are received, process p updates its load and returns the first received task. If no tasks were received or no task requests were emitted, processor p signal its termination and exit the RID algorithm.

Minor modification

The notion of processor neighbourhood plays a fundamental role in this algorithm. It defines the speed of diffusion of the load across the network of processors and partakes in the stopping criterion. Indeed, the local completion of a processor p, happening in algorithm 12, is based on the criterion that no more work can be or has been requested from any neighbours $q \in \psi_p$. Unfortunately, this criterion does ensure that a neighbour processor q will not subsequently receive a significant load afterwards.

Indeed, whenever the task costs are highly variable and the network is large with respect to the neighbourhood, it is most probable to observe a neighbourhood of processors $\psi_\star$ to finish locally their tasks while the remaining processors are still significantly loaded (Fig. 5.1). These overloaded processors may be in the process of computing expensive tasks and therefore can not answer to pending tasks requests. Thus the *inner* part of

Algorithm 12 Operation applied by processor p before computing a task t_i, if such a task is available.

```
if A task t_i is locally available then
  return t_i
end if
while j_p > 0 do
  Answer pending task requests
  wait for a task request to be answered
  j_p = j_p − 1
  if Received k new jobs then
    Update task queue and l_p
    return t_i
  end if
end while
Send done signal to neighbours and exit
```

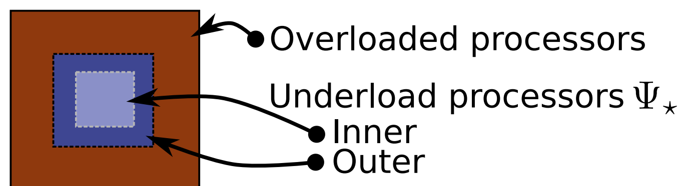


Figure 5.1.: Illustration of a problematic load distribution.

the neighbourhood $\psi_\star$ could detect that neighbours have no more load to share and their stopping criterion would then be met. However, when the overloaded processors would eventually diffuse their load to the *outer* part of $\psi_\star$, the processors having previously reached the local completion would stand idle.

In order to address this issue, a processor p identifying a RID local completion shifts to a *sender initiated diffusion* (SID) state. This requires a modification of algorithm 12 that requires p to memorise its new state and signal it to its neighbours. Once in the SID state, processor p has to wait for tasks that could be sent by neighbours. Its local completion is then reached upon reception of the SID state notification from all his neighbours $q \in \psi_p$.

Algorithm 11 is modified consequently to specialise its behaviour in function of its neighbour state. Indeed, processors in RID continues to send tasks requests to overloaded neighbours in state RID, but pushe tasks to underloaded neighbours in SID state.

This two-phases variant of the original algorithm delays the local completion of a processor by requiring that all its neighbours $q \in \psi_p$ have no more tasks to share and no pending tasks requests. This added termination criterion reduces the risk of observing the problematic scenario previously identified.

5.1.2. Implementation in HOGAN

The enhanced RID algorithm is implemented in HOGAN using the standard MPI and represents the highest layer of parallelism as illustrated in figure 5.2. Therefore, several independent tasks identifying ML estimations or MCMC samplings can be managed within HOGAN to be executed on high performance computers.

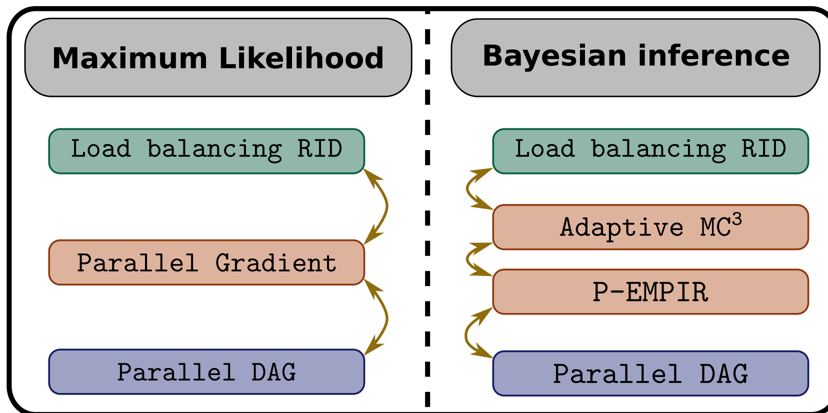


Figure 5.2.: Illustration of the combined layers of parallelism for ML and Bayesian inference.

In order to fully benefit from these highly parallel architectures, as well as simpler ones, HOGAN supports multiple kinds of processor neighbourhood that are now described. Following this description, a small illustrative benchmark is presented where the perfor-

mance of the enhanced RID is analysed. Finally, the major limitation of the presented approach is identified and potential solutions to this problem are proposed.

Processor neighbourhood

The diffusion of the load using the RID algorithm is highly dependent of the processors neighbourhoods considered. Indeed, this neighbourhood defines the worst-case, or maximal, distance that tasks have to travel from an overloaded processor to some underloaded other processor. Ideally, if the processors were fully connected, this maximal distance would be of one. However, given the nature of the algorithm, this approach would cause the network to be overflowed with load status messages and job requests.

In the context of `HOGAN`, the communication overhead inherent to large neighbourhoods should not have a significant impact on the performance given that the computational cost of tasks is dominant. Therefore, five different torus-based neighbourhoods are available and can be chosen according to the resources employed for the analysis. These neighbourhoods are now shortly described and their associated maximal distance is illustrated for an analysis using $P = 8192$ processors.

- `1D`-torus: each processor has two neighbours and the maximal distance is $P/2 = 4096$.
- `2D`-torus: each processor has four neighbours, two in x and y dimensions that are chosen to be as close as possible to $\sqrt{P}$. For $P = 8192$, the torus has $x = 128$ and $y = 64$, therefore the maximum distance is $x/2 + y/2 = 96$.
- `3D`-torus: each processor has six neighbours. The z dimension is defined as the number of cores on a processor, e.g $z = 16$, and the x, y dimension are defined as in the `2D`-torus. In this example, there are thus $P/z = 512$ plans with $x = 32, y = 16$ processors wich give a maximal distance of $x/2 + y/2 + z/2 = 32$.
- `5D`-torus: this neighbourhood is specifically implemented for the Blue Gene/Q[1] supercomputer that has an interconnection network designed as a `5D`-torus of dimension $2 \times 4 \times 4 \times 4 \times 4$. This network interconnects 16 core processors that communicate according to a `2D`-torus in `HOGAN`. The coupling of both torus results in each processor having eleven neighbours and a maximal distance of $1 + 2 + 2 + 2 + 2 + 2 + 2 = 13$.

This variety of neighbourhoods enables the use of this algorithm in various settings. For instance, during the elaboration of a database, hundreds of thousands of tasks using a single processor would benefit from the `5D`-torus on supercomputers such as the Blue Gene/Q. Conversely, for a thorough analysis with the branch-site model on a large model, hypotheses would be computed using parallel DAG and gradient computations and therefore the use of a `2D` or `3D`-torus would be more appropriate on an ordinary cluster.

[1] `http://bluegene.epfl.ch/`

Experiments on simulated tasks

A small illustrative experiment was conducted by simulating the execution of synthetic tasks over P processors. These n_t tasks were generated as to present different scenarios of load imbalance. Each processor generated n_t/P tasks having computational costs drawn from the normal distribution $\mathcal{N}(\mu_p, \sigma_a^2)$. The mean μ_p of each processor distribution was defined by another normal distribution $\mu_p \sim \mathcal{N}(\mu, \sigma_b^2)$ such that σ_b^2 controls the imbalance over processors.

For these experiments, the following values were used

$$n_t = 6400, \quad \mu = 100, \quad \sigma_a^2 = 15 \quad \text{and} \quad \sigma_b^2 = \{10, 30\}.$$

The fictive computations identified by these parameters were replicated five times and each of the replica was ran with the original and enhanced RID algorithms on 16, 32 and 64 processors. Furthermore, each of these scenario was tested with `1D-`, `2D-` and `3D`-torus[2] neighbourhoods and with two different estimations of the task cost: a task t_i was either approximated as the mean of its generative distribution, $\mathcal{C}(t_i) = \mu_p$, or a unit cost, $\mathcal{C}(t_i) = 1$.

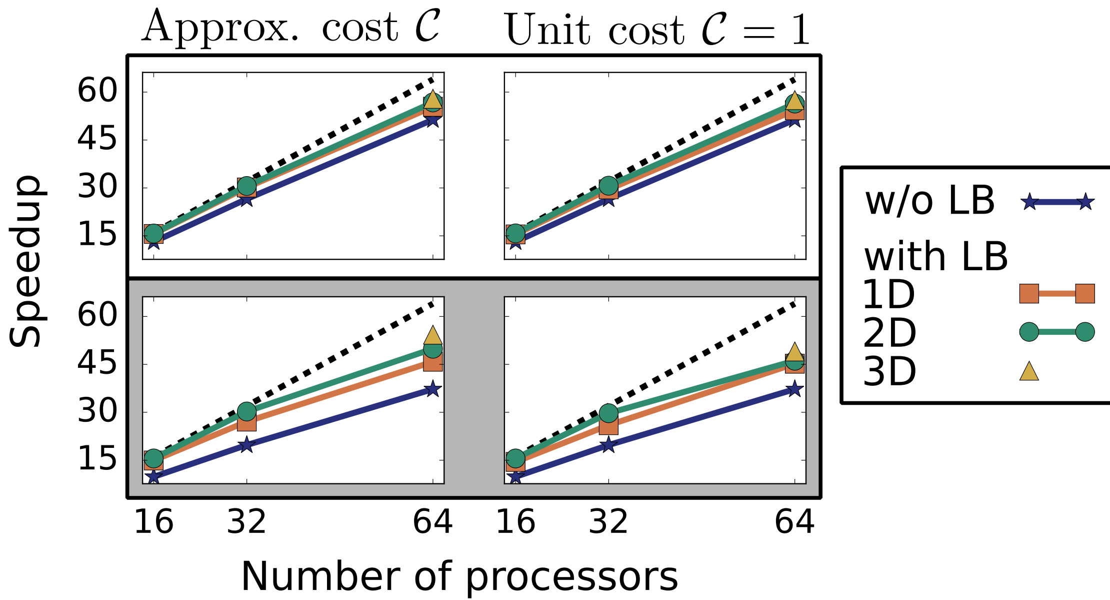


Figure 5.3.: Illustration of the enhanced RID performance on artifical tasks. The blue line represents the speedup of a run without load balancing, the black dotted is the ideal speedup and the other lines are runs with RID using different neighbourhoods. Top and bottom figures had artificial tasks simulated with $\sigma_b^2 = 10$ and $\sigma_b^2 = 30$, respectively.

[2]Only 64 processors could benefit from a `3D`-torus.

The results of these experiments is synthesised in figure 5.3. As expected, the performance of the algorithm was highly dependent on the load imbalance defined by σ_b^2. Indeed, when a small imbalance was simulated the speedup was nearly linear, while it decreased when the imbalance increased. The effect of the imbalance was furthermore coupled with the decrease of load per processor associated with the increase of the processor number P. A lesser load per processor implies that the unbalance affects more strongly the speedup and that the algorithm takes less *time* to diffuse the load.

However, even in the worst cases (figure 5.3d), the RID algorithm significantly improved the speedup when compared to an execution without any load balancing. The performance of the algorithm was positively impacted by the knowledge of good approximations of the task costs. In the context of `HOGAN`, such approximations can be obtained by considering the relative difference of the datasets size that directly impact the task computational cost. The use of large neighbourhoods also improved the performance of the RID algorithm. Furthermore, these faint improvements due to the neighbourhood should increase on executions having several hundreds of processors.

Finally, comparative measures with the original RID algorithm showed that its enhanced version was improving the speedup of upto 5% on highly imbalanced scenarios executed with 64 processors. On the simpler scenarios, no significant difference was observed.

Limitation

The major limitation residing in this implementation of RID is that the amount of processors dedicated to each task must be identical. For example, during a ML analysis, if four processors are dedicated to estimate the gradients in parallel, then all the tasks have to use four processors for this parallel method.

This approach can be problematic when analyses are conducted on datasets having huge variation in sizes. Indeed, as discussed in chapters 2 and 3, the efficiency of the parallel methods depends on the size of the datasets and thus dedicating several processors to a small dataset may be a waste of computational resources. Conversely, large datasets may require several processors to be analysed within the limit of time imparted.

Therefore, in the current state of `HOGAN`, users are responsible for making a sensible choice with respect to the parallel settings as well as the datasets analysed. However, more evolved strategy could be envisioned to tackle this challenge. Using approximation of an analysis cost, based on the dataset size or on execution times measured on the DAG, the parallel setting could be automatically adapted to fit the problem at hand.

For instance, depending on the initial distribution of the dataset sizes, the computational resources could be decomposed so as to provide efficient *solvers* for each category of computational tasks. Another approach could be to dynamically adapt the assignment of computational resources during an execution as a function of the measured execution times of the DAG nodes. Such approaches would require far more complex and evolved scheduling methods than the RID algorithm and would thus represent a major challenge that could however significantly improve the performance of `HOGAN` on large scale analyses.

5.2. Building a database of coevolution

Originally, the need for an asynchronous and distributed load balancing algorithm was motivated by the ambitious project[3] of building a large database containing results of coevolution analyses conducted with the model presented in section 1.3.5. Prior to this model, analyses of coevolution on molecular data were mainly done using correlation measures on molecular sequences. By modelling coevolution as an evolutionary process on a phylogeny, this model enabled the study of this phenomena in a more realistic context.

However, when compared with correlation analyses, this model has a fairly high computational cost that may prove prohibitive to biologists used to the correlations-based methods. This expensive computational cost comes from the large amount of distinct ML estimations that must be conducted to fully analyse a dataset composed of aligned sequences of size N. Indeed, the `Coev` model considers subsequently all the $N^2/2$ possible pairs of positions and this for all the observed profile of coevolution that may amount to M (Eq. (1.9)) in the worst case. For instance a sequence of 200 nucleotides could represent, in the worst case, more that $M \cdot N^2/2 \approx 10,000$ ML estimations for the `Coev` model and as many for the null model.

Therefore, an effort was made to elaborate a web-service [28] making available the analyses of `Coev` to anyone through the infrastructure of the *Swiss Institute of Bioinformatics*. While enabling these analyses on a dataset basis represented a significant progress, it failed to deliver the required tool to study coevolution signals detected with this model on a larger biological scale. Therefore, a more significant effort was made to produce a database of coevolution covering the same dataset as the `Selectome` [98, 87] database.

This ongoing adventure, started two years ago, is shortly presented in this section. The pipeline of software is first presented and followed by a description of one of the tool designed for its automation. Finally, the current status of this large scale analysis is summarised.

5.2.1. Pipeline

The pipeline required to build the `Coev` database is detailed in figure 5.4. The first step consisted in the extraction and the pre-processing of the datasets from the `Selectome` database. After extraction, the two extremes of the dataset size, namely the tiny and enormous sequences, were filtered as well as the ambiguous positions of the sequences. The pre-processing also determined the coevolution profiles observable so as to provide the list of ML estimations to conduct.

This list was transmitted to a dedicated server running an Automated Job Manager (AJM). The AJM took care of preparing jobs representing a set of ML estimations, or tasks, that were then periodically submitted on a cluster in order for ML estimations to be computed using the coupling of an optimised version of the original `Coev` implemen-

[3] This work results from a collaboration with the original author of `Coev`, L. Dib.

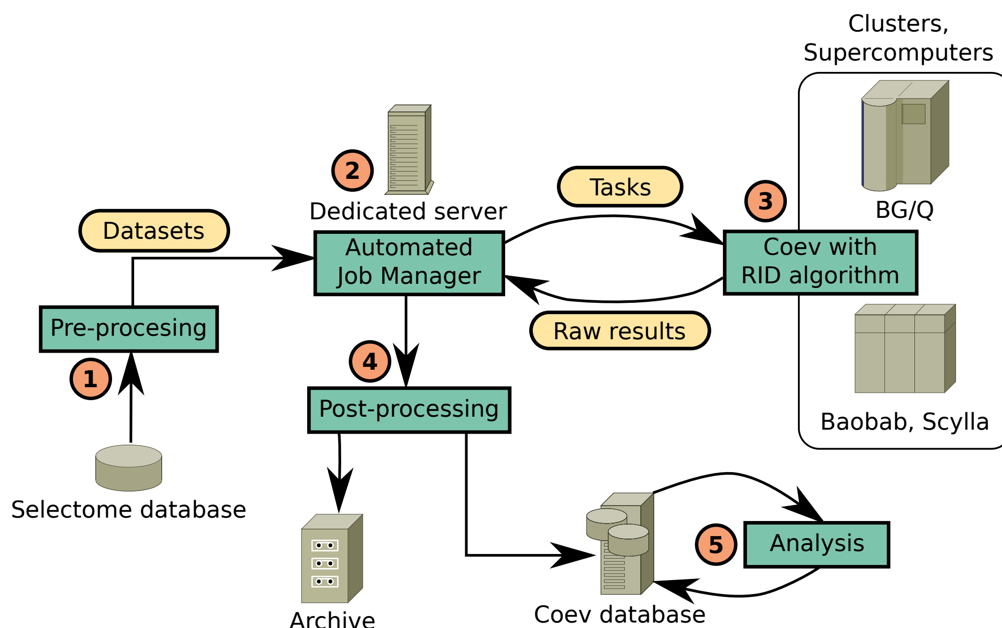


Figure 5.4.: Illustration of the pipeline implemented for populating the `Coev` database. The five key operation are detailed in the green boxes numbered in the red circle.

tation with the RID algorithm. The execution of these jobs were then monitored by the AJM and, once done, their results were retrieved by the AJM.

After having verified the correct execution of the jobs, the AJM launched the post-processing of the raw results that consisted in mapping back the analysed positions of the filtered sequences with their position in the original sequence. These post-processed results were then archived on a tape-based storage server and inserted in a relational database installed on a server dedicated for the `Coev` database.

Automated Job Manager

The AJM was specially designed for the `Coev` database project and without this software the database, or a large part of this thesis, would not be existing today. Indeed, this software automates the whole lifetime of a job that encompasses numerous steps: its submission, tracking, retrieval, verification and archiving. Assuming that manually managing a job would take 15 minutes, the $3,000$ jobs executed over a period of two years would represent more than 5 months of full-time work. This estimation is furthermore unrealistic since it underestimates the complexity of the task and does not take into account human-related factor such as vacations, illness or errors.

The AJM was thus conceived[4] to provide a safe and stable service that would manage

[4] The development phase took two weeks and the operational launch took one week.

a large amount of jobs over a long period of time. For that matter, this solution had to maintain the current state of this tedious process in persistent memory so as to inform the end-user and provide the required means of managing the various computational resources as well as their job assignments.

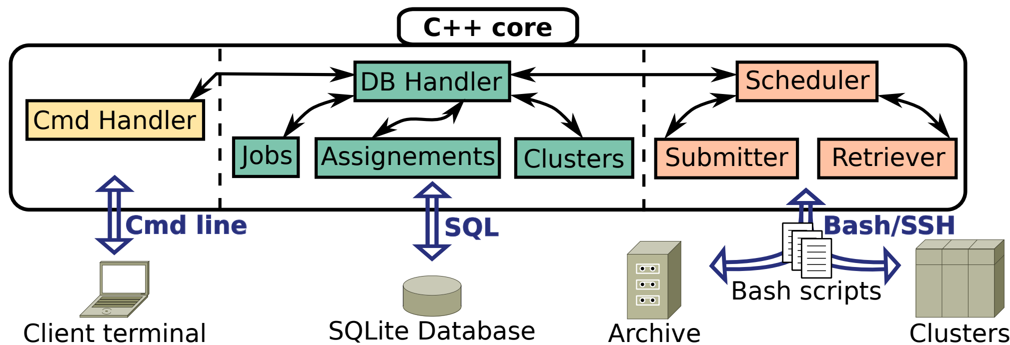


Figure 5.5.: Illustration of the flow of information in the Automated Job Manager software.

To meet these criterion the AJM was conceived using a mixture of C++, SQL and bash scripts, as illustrated in figure 5.5. The core C++ software manages three key aspects: the interactions with the user, the persistent data and the execution of jobs. The persistent data, maintained in a SQLite database, acts as the pivotal element of the AJM by enabling information to be exchanged between the end-user and the job scheduler.

The end-user interacts with the software through a command line interface that enables the state of the pending or executed jobs to be queried as well as the registration of cluster and job assignments. These assignments define at which rate and on which clusters jobs must be submitted. This aspect is critical for the safe use of the computational resources: strategies had to be employed to ensure that failing jobs or communication losses with clusters were correctly managed and reported to the end-user.

These strategies are enforced by the scheduler that creates and manages jobs according to the assignments specified by the user. In function of a job state, the scheduler triggers the main operations applied to jobs, such as its submission or retrieval. These operations are performed by bash scripts for two reasons: it enables these operations to be easily adapted (e.g. pre/post-processing) and it facilitates the interaction with the clusters by calling simple remote scripts specialised for them.

5.2.2. Current state of `Coev` database

The presented pipeline was active for the last two years and resulted in a first version of the `Coev` database. This enterprise can be synthesised by the following numbers:

- **3** clusters were used: two medium sized clusters periodically and a supercomputer (BlueGene/Q).

- **20%** of the BlueGene/Q was dedicated to `Coev` during these two years.
- **3,000+** jobs were submitted and monitored 24/7, equating to more than 4 jobs per days.
- **8,200+** proteins, or aligned sequences, were fully analysed.
- **500,000+** strong signals of coevolution were detected.
- **11 billions** of hypotheses testing were conducted, each of them requiring two ML estimations.
- **1,2 Terabyte** of data are required to store the resulting database.

This first version of the database is currently analysed as to extract a summary for each proteins of the dominant coevolution signal detected. This summary could provide an insight on the distribution of coevolution signals with respect to their profiles, the gene functionalities or other available gene annotations. Such information could already be humanly exploitable and would deserve to be shared. Then a second, more complete, analysis could be conducted on the whole database using data mining strategies so as to detect potential relations or anomalies expressed in this unprecedented amount of information on coevolution.

5.3. Summary

In this chapter, the problematic of executing large scale analyses is studied. A model-generic strategy designed to exploit HPC infrastructures and based on an existing load balancing algorithm is detailed. Finally, a large scale application of this strategy resulting in the creation of a coevolution database is presented.

Model-generic

The strategy used to profit from HPC infrastructure is based on an existing dynamic, asynchronous and distributed load balancing algorithm known as RID. This algorithm is slightly adapted and implemented as follow:

- Enhancement of the RID algorithm to help the diffusion of computational tasks on supercomputers.
- Implementation of several processor neighbourhoods.

This implementation enables HPC infrastructure to be employed for a broad range of scenarios going from a large amount of mono-processor analyses and ending with a small amount of analyses using a large amount of processors. Indeed, the integration of this load balancing strategy is designed so as to enable the joint use of the other parallel methods implemented in `HOGAN` (DAG, ML and MCMC).

Model-specific

A small performance study is presented for the validation of the enhanced RID algorithm. Then the laborious construction of a large database of coevolution that motivated the implementation of this load balancing strategy is detailed:

- The pipeline required for the creation of the `Coev` database is documented.
- A crucial tool developed for the automation of the gigantic amount of operations inherent to the pipeline is shortly presented.
- The current state of the `Coev` database, that indexes more than 11 billions results, and its perspectives are synthesised.

6. Conclusion

Statistical inference on nowadays evolutionary biology models is hindered by their increasing complexity as well as the ever growing amount of available data. In chapter 1, some models encountering such limitations were presented and the numerous existing model-specific or model-generic software related to such a problematic were listed. From this state-of-the-art, the need for a generic framework for statistical modelling emerged.

Five properties desirable for such a framework were identified. First, it had to be **modular** and **flexible** to enable the integration of state-of-the-art models and methods. Then, in order to deal with complex models having computationally expensive likelihood functions or large datasets or both, the framework had to be **efficient** and present **support for HPC** infrastructure. Finally, given that a tool without users serves no purpose, it had to be **user-friendly** for both end-user (e.g. biologists) and model developers (e.g. bioinformaticians or biostatisticians).

This final chapter assesses if `HOGAN` incorporates these key features. While several key algorithms implemented in `HOGAN` were thoroughly detailed in the previous chapters, this targeted description fails to give the whole picture of the framework. For that matter, `HOGAN` is shortly described from a *software engineering* point of view so as to provide a solid support to discuss each of the five properties. Following this discussion, this thesis part is concluded by an overview of the potential enhancements and future applications of `HOGAN`.

6.1. `HOGAN` in a nutshell

The simplified class diagram shown in figure 6.1 synthesises the key elements of `HOGAN` and their relations. The pivotal element of the framework is contained in the *Statistical model* package that defines a model having an implementation of the *Likelihood* interface. The *Likelihood* class extensively relies on the functionality and classes provided in the *DAG* package that encompasses the required elements to define a likelihood as a directed task graph and schedule its computation. These two packages contain the implementations of the features discussed in chapter 2.

The *ML* and *Sampler* packages contains the implementation of the sequential and parallel methods described in chapter 3 and 4, respectively. Each of these two packages have an interface that respectively defines the critical class for the *MCMC* or the *Optimiser* methods. Furthermore they extensively uses the *PLU Utils* package that enable the observed execution time of the DAG to be extracted and exploited to enhance the method of finite differences or to enable the automated creation of parameter blocks.

The *LB RID* package enables `HOGAN` to be used for large scale analyses and was described in the chapter 5. The coupling of all the different parallel methods that can be

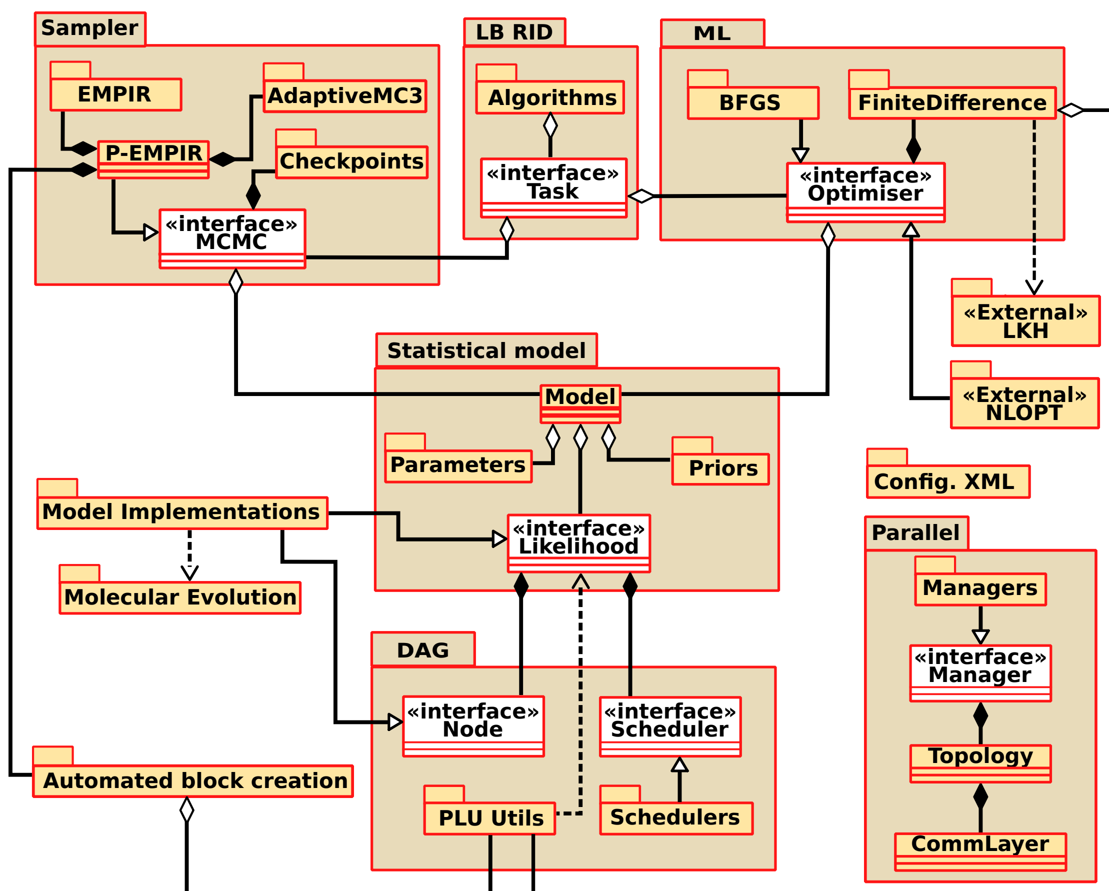


Figure 6.1.: Simplified UML class diagram of the main components of HOGAN. Important pure virtual classes (*interfaces*) are shown in white. Packages, represented as folders, encompass a set of classes and are named after their functionality.

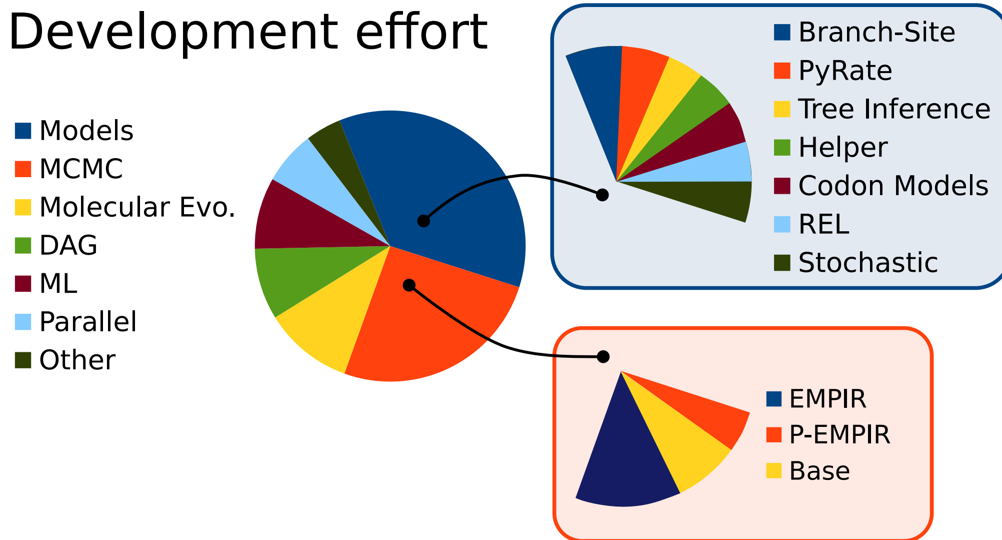


Figure 6.2.: Distribution of the 47'000 physical Source Lines Of Code (SLOC) composing `HOGAN`. The SLOC was generated using David A. Wheeler's `SLOCCount`.

employed in such analyses is managed by classes in the *Parallel* package. The *Topology* class organises the multiple communication layers (*CommLayer*) such that each processor knows its exact role and neighbours in each of the parallel methods.

Finally, the *Model Implementations* and *Molecular Evolution* packages contains two crucial elements of `HOGAN`, yet their existence was solely sporadically discussed during the previous chapters. These two packages represent a large part of the development effort, as illustrated in figure 6.2, and contain all the implemented models and evolutionary biology related functionalities. The *Molecular Evolution* package encompasses all the helper functions such as the data loaders for molecular sequences and phylogenetic trees, the definition of evolutionary models (substitution matrices), the data structures for phylogenetic trees and their dedicated tree proposals.

The *Model Implementations* package contains the implementations of the models used to illustrate `HOGAN`'s functionalities and to conduct other experiments. The following models have been implemented using the DAG-based representation in `HOGAN`:

- Two variants of the `PyRate` model [114].
- Several model for the detection of the positive selection on protein coding genes:
  - Five variants of the codon-substitutions site model [137].
  - The codon-substitution branch-site model [138].
  - The Random Effect Lineage (REL) branch-site model [97].
  - The stochastic branch-site model [49].

- The model for the phylogenetic tree inference with simple nucleotide and codons-substitution models [33, 76].

Furthermore, to ease the use of all these models, *Helper* classes are readily available and provide initial configuration for the *Parameter* and *Prior* objects as well as the crucial model-specific ordering of the parameters and default parameter blocks discussed in chapter 3 and 4.

Flexibility and modularity

The flexibility and modularity of the framework can be directly assessed from the class diagram (Fig. 6.1). Any statistical models implementing the *Likelihood* interface and using the directed task graph representation described in the *DAG* package are able to benefit from all the functionalities of the framework.

In addition of being model-generic, `HOGAN` is also *method-generic*. Indeed, several interfaces enable the addition of new DAG schedulers or new MCMC and optimization methods. If such new functionalities would be based on parallel algorithms, their integration in the parallel hierarchy were readily specifiable using the *Manager* interface.

This design enforced by the use of clear interfaces ensures that any existing functionalities or models will benefit, or be beneficial, to novel additions in the framework. Therefore, this modular design enables `HOGAN` to provide a flexible framework having the power of integrating the next generation of models and methods.

Efficiency

The efficiency of the methods designed and implemented in `HOGAN` have been detailed in length in their respective chapter as well as summarised at their end. These methods can be outlined by first considering the likelihood evaluations that are advantageously represented as directed task graphs such as to benefit from PLUs and parallel DAG scheduling.

ML methods benefit from this DAG representation since it permits the novel approach of optimising the gradient computation in sequential and parallel by using an existing state-of-the-art heuristic for the widely studied Traveller Salesman Problem. These model-generic strategies were applied to the branch-site model identifying positive selection and outperformed a state-of-the-art software `FastCodeML` [127] by upto three orders of magnitude.

Bayesian inference benefits from the design and implementation of a novel family of adaptive proposals, `EMPIR`, and their synergistic coupling with a parallel M-H algorithm, entitled `P-EMPIR`. These innovative model-generic approaches significantly improved the sampling efficiency of the highly dimensional posterior distribution of several complex models. When compared to `MrBayes` [106], a widely used state-of-the-art software for Bayesian inference of phylogeny, `HOGAN` was shown to be upto 20 times more efficient; results that was furthermore identified as having the potential to be doubled by the automated creation of parameter blocks.

Therefore, while being model-generic, HOGAN proved to be more efficient than two model-specific state-of-the-art implementations. The magnitude of the performance gains observed during the experiments conducted opens the way for novel analyses of molecular data that were deemed intractable until now. Finally, on top of these model-generic approaches, model-specific strategies optimising the ordering of gradient computations as well as the creation of parameter blocks, were derived for the phylogeny based models. These inexpensive approaches enabled good solutions to be found for these complex combinatorial problems at an insignificant cost, and thus further augmented the efficiency of HOGAN.

HPC support

Large scale analyses, such as the creation of the Coev database presented in chapter 5, are possible in HOGAN thanks to its implementation of a asynchronous and distributed load balancing algorithm. The application of statistical analyses to a large amount of datasets does not represent the only potential use of HPC architecture for HOGAN. Indeed, by combining several layers of parallelism, HOGAN can take advantage of supercomputers having several hundreds of processors for the analysis of a single dataset.

For instance, Bayesian inference on a large dataset could be conducted with more than 2,000 processors by combining the replication of 4 independent samplers using each 4 parallel MC^3 processes, themselves having each P-EMPIR running with 32 parallel likelihood *evaluators* with 4 dedicated processors. Similarly, ML-based inference could use more than 1,000 processors by replicating 5 analyses of 2 hypotheses. These 10 independent ML estimations could be conducted using 32 parallel gradient estimators each evaluating the likelihood using 4 processors.

User friendliness

Two categories of potential users were identified: end-users and developers. The first category, end-users, can readily configure runs of HOGAN through the XML configuration files (package *Config. XML*) that could as well provide an advantageous interface for a graphical user interface. While some fields of these files are mandatory, several default values can be automatically loaded by *Helper* classes and thus, users are not obligated to set everything by themselves. For Bayesian inference, end-users benefit from the auto-tuning of proposals that P-EMPIR provides as well as a checkpoint system that enables the restart of the MCMC sampler with the adapted proposals.

Developers can readily augment the framework with models or methods that implement the existing interfaces. A significant effort was dedicated to ease the implementation of models and to provide the necessary means to easily design efficient likelihood computations. This part is ensured by the data structures and tools provided in the *DAG* package. Furthermore, developers of phylogenetic based models dispose of several key functionalities that are provided in the *Molecular Evolution* package. The overview of the development effort in code line (Fig. 6.2) reveals that the effort dedicated to the implementation of a model remains fairly small compared to the large amount of

functionalities made available to the model-developer.

In conclusion, both categories of users have at their disposal several tools that facilitate their interaction with HOGAN. Therefore, the framework has the potential to be user friendly; however it is only through the practice that this property will be duly assessed and subsequently refined.

6.2. Future perspectives

In conclusion, HOGAN is a promising candidate for filling the existing need, at least in evolutionary biology, for a generic framework for statistical modelling and is already exploitable for the study of large datasets deemed intractable until now. Existing models not yet implemented, such as the relaxed molecular clock models [79] used to date branching events on phylogenetic trees and known for their highly correlated parameters, would also benefit from the efficient sampling and advanced proposals implemented in P-EMPIR. More importantly, the next generation of models in evolutionary biology is predicted to rely on the integration of several levels of macro-evolutionary processes [78] which would imply to delegate the selection of models to the MCMC sampler by using transdimensional Markov chains [117]. The augmented parameter space induced by these approaches would further justify the need of advanced samplers such as HOGAN.

Indeed, novel functionalities such as the automated and adaptive creation of parameter blocks would play a crucial role in making such complex models tractable. Change in the model structure, and thus in the directed task graph representing the likelihood, induced by transdimensional moves could be reflected in parameter blocks and proposals. These adaptations would enable a significant improvement in the sampling efficiency of these model posterior distributions. However, some aspects of HOGAN remain to be polished to reach the level of maturity required for such models. Indeed, the automated creation of blocks would benefit from the integration of the parameter correlations observed in the partial covariance matrices and from a more detailed performance model defining the impact of parameter block size. Another refinable functionality is the strategy employed for large scale analyses which lacks flexibility with respect to tasks having highly uneven computational costs and thus could benefit from a more evolved approach that would allocate intelligently the computational resources.

Both of these problems, as well as the scheduling of parallel likelihood evaluations and the scheduling of gradient approximation, can be formalised as complex combinatorial or multi-objective optimization problems. Solutions for these problems were obtained using several different sequential heuristic methods and, apart for the sequential scheduling of gradient approximations that used a state-of-the-art method [57], these heuristics have significant room for improvement. Instead of improving them separately, a more ambitious approach would be to consider the integration of a parallel metaheuristic method [16, 3] that could produce high-quality solutions for all these optimization problems and benefit from the inherent parallelism of HOGAN. Furthermore, this approach would have several added advantages: the reduction of the reliance of HOGAN on external libraries, a readily available tool for additional optimization opportunities and an effi-

cient strategy to define good starting parameter values prior to an analysis with ML or MCMC method.

In addition to these improvements on the core of `HOGAN`, recent advances in computational statistics could prove to be invaluable. Indeed, while `HOGAN` enables the analyses of large datasets, even larger ones are existing or being assembled [58, 122] and therefore, `HOGAN` could create a bridge between these practical challenges and the ongoing research efforts to provide Bayesian inference strategies for Big Data [111, 132]. Ongoing efforts to provide more powerful proposal kernels that explore complex parameter spaces, such as the Langevin diffusion-based kernels [9, 43] for continuous parameters or the phylogenetic tree based proposals [17, 2] could also serve as a solid basis for further developments of `EMPIR` and `P-EMPIR`. Lastly, approximation of the posterior distribution obtained trough the adaptive phase of `EMPIR` could be coupled to delayed-acceptance scheme so as to avoid the evaluation of presumably unlikely parameter values [53] or could be coupled with more advanced approximation techniques [113] derived from machine learning.

In conclusion, `HOGAN` and the numerous strategies designed to improve its efficiency provide a significant improvement when compared to the existing software of statistical inference in evolutionary biology. Moreover, this flexible and HPC-ready framework may prove a timely incubator for the next generation of evolutionary biology models and statistical methods by enabling them to meet on a common ground.

Bibliography

[1] Aberer, Andre J., Kassian Kobert, and Alexandros Stamatakis. "ExaBayes: Massively Parallel Bayesian Tree Inference for the Whole-Genome Era." Molecular Biology and Evolution 31, no. 10 (October 2014): 2553–56. doi:10.1093/molbev/msu236.

[2] Aberer, Andre J., Alexandros Stamatakis, and Fredrik Ronquist. 2015. "An Efficient Independence Sampler for Updating Branches in Bayesian Markov Chain Monte Carlo Sampling of Phylogenetic Trees." Systematic Biology, July, syv051. doi:10.1093/sysbio/syv051.

[3] Alba, Enrique, Gabriel Luque, and Sergio Nesmachnow. 2013. "Parallel Metaheuristics: Recent Advances and New Trends." International Transactions in Operational Research 20 (1): 1–48. doi:10.1111/j.1475-3995.2012.00862.x.

[4] Altekar, Gautam, Sandhya Dwarkadas, John P. Huelsenbeck, and Fredrik Ronquist. 2004. "Parallel Metropolis Coupled Markov Chain Monte Carlo for Bayesian Phylogenetic Inference." Bioinformatics 20 (3): 407–15. doi:10.1093/bioinformatics/btg427.

[5] Andrieu, Christophe, and Johannes Thoms. 2008. "A Tutorial on Adaptive MCMC." Statistics and Computing 18 (4): 343–73. doi:10.1007/s11222-008-9110-y.

[6] Andrieu, Christophe, Arnaud Doucet, and Roman Holenstein. 2010. "Particle Markov Chain Monte Carlo Methods." Journal of the Royal Statistical Society: Series B (Statistical Methodology) 72 (3): 269–342. doi:10.1111/j.1467-9868.2009.00736.x.

[7] Applegate, David L, Robert E Bixby, Vasek Chvatal, and William J Cook. 2011. The Traveling Salesman Problem: A Computational Study. Princeton university press.

[8] Arora, Sanjeev. 1998. "Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and Other Geometric Problems." J. ACM 45 (5): 753–82. doi:10.1145/290179.290180.

[9] Atchade, Yves F. 2006. "An Adaptive Version for the Metropolis Adjusted Langevin Algorithm with a Truncated Drift." Methodology and Computing in Applied Probability 8 (2): 235–54. doi:10.1007/s11009-006-8550-0.

[10] Ayres, Daniel L., Aaron Darling, Derrick J. Zwickl, Peter Beerli, Mark T. Holder, Paul O. Lewis, John P. Huelsenbeck, et al. "BEAGLE: An Application Programming Interface and High-Performance Computing Library for Statistical Phylogenetics." Systematic Biology 61, no. 1 (January 1, 2012): 170–73. doi:10.1093/sysbio/syr100.

[11] Bader, David A., Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Graph Partitioning and Graph Clustering. American Mathematical Soc., 2013.

[12] Baussand, Julie, and Alessandra Carbone. "A Combinatorial Approach to Detect Coevolved Amino Acid Networks in Protein Families of Variable Divergence." PLoS Comput Biol 5, no. 9 (September 4, 2009): e1000488. doi:10.1371/journal.pcbi.1000488.

[13] Beaulieu, Jeremy M., Dwueng-Chwuan Jhwueng, Carl Boettiger, and Brian C. O'Meara. "Modeling Stabilizing Selection: Expanding the Ornstein–Uhlenbeck Model of Adaptive Evolution." Evolution 66, no. 8 (August 1, 2012): 2369–83. doi:10.1111/j.1558-5646.2012.01619.x.

[14] Beiko, Robert G., Jonathan M. Keith, Timothy J. Harlow, and Mark A. Ragan. "Searching for Convergence in Phylogenetic Markov Chain Monte Carlo." Systematic Biology 55, no. 4 (August 1, 2006): 553–65. doi:10.1080/10635150600812544.

[15] Bektas, Tolga. "The Multiple Traveling Salesman Problem: An Overview of Formulations and Solution Procedures." Omega 34, no. 3 (June 2006): 209–19. doi:10.1016/j.omega.2004.10.004.

[16] Bianchi, Leonora, Marco Dorigo, Luca Maria Gambardella, and Walter J. Gutjahr. 2008. "A Survey on Metaheuristics for Stochastic Combinatorial Optimization." Natural Computing 8 (2): 239–87. doi:10.1007/s11047-008-9098-4.

[17] Bouchard-Cote, Alexandre, Sriram Sankararaman, and Michael I. Jordan. 2012. "Phylogenetic Inference via Sequential Monte Carlo." Systematic Biology 61 (4): 579–93. doi:10.1093/sysbio/syr131.

[18] Brockwell, A. E. 2006. "Parallel Markov Chain Monte Carlo Simulation by Pre-Fetching." Journal of Computational and Graphical Statistics 15 (1): 246–61. doi:10.1198/106186006X100579.

[19] Brooks, Stephen P., and Andrew Gelman. "General Methods for Monitoring Convergence of Iterative Simulations." Journal of Computational and Graphical Statistics 7, no. 4 (December 1, 1998): 434–55. doi:10.2307/1390675.

[20] Buluç, Aydin, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. "Recent Advances in Graph Partitioning." CoRR abs/1311.3144 (2013). http://arxiv.org/abs/1311.3144.

[21] Cappe, O., S.J. Godsill, and E. Moulines. 2007. "An Overview of Existing Methods and Recent Advances in Sequential Monte Carlo." Proceedings of the IEEE 95 (5): 899–924. doi:10.1109/JPROC.2007.893250.

[22] Chamberlain, Bradford L, and others. 1998. "Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations." University of Washington Technical Report UW-CSE-98-10 3.

[23] Dauvergne, Benjamin, and Laurent Hascoët. "The Data-Flow Equations of Checkpointing in Reverse Automatic Differentiation." In International Conference on Computational Science, 566–573. Springer, 2006.

[24] Darriba, D., A. Aberer, T. Flouri, T.A. Heath, F. Izquierdo-Carrasco, and A. Stamatakis. "Boosting the Performance of Bayesian Divergence Time Estimation with the Phylogenetic Likelihood Library." In Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International, 539–48, 2013. doi:10.1109/IPDPSW.2013.267.

[25] Di Battista, Giuseppe, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. 1994. "Algorithms for Drawing Graphs: An Annotated Bibliography." Comput. Geom. Theory Appl. 4 (5): 235–82. doi:10.1016/0925-7721(94)00014-X.

[26] Dib, Linda, and Alessandra Carbone. "CLAG: An Unsupervised Non Hierarchical Clustering Algorithm Handling Biological Data." BMC Bioinformatics 13 (2012): 194. doi:10.1186/1471-2105-13-194.

[27] Dib, Linda, Daniele Silvestro, and Nicolas Salamin. "Evolutionary Footprint of Coevolving Positions in Genes." Bioinformatics 30, no. 9 (May 1, 2014): 1241–49. doi:10.1093/bioinformatics/btu012.

[28] Dib, Linda, Xavier Meyer, Panu Artimo, Vassilios Ioannidis, Heinz Stockinger, and Nicolas Salamin. "Coev-Web: A Web Platform Designed to Simulate and Evaluate Coevolving Positions along a Phylogenetic Tree." BMC Bioinformatics 16 (2015): 394. doi:10.1186/s12859-015-0785-8.

[29] Duane, Simon, A.D. Kennedy, Brian J. Pendleton, and Duncan Roweth. 1987. "Hybrid Monte Carlo." Physics Letters B 195 (2): 216–22. doi:10.1016/0370-2693(87)91197-X.

[30] Drummond, Alexei J, Simon Y. W Ho, Matthew J Phillips, and Andrew Rambaut. "Relaxed Phylogenetics and Dating with Confidence." PLoS Biol 4, no. 5 (March 14, 2006): e88. doi:10.1371/journal.pbio.0040088.

[31] Drummond, Alexei J., Marc A. Suchard, Dong Xie, and Andrew Rambaut. "Bayesian Phylogenetics with BEAUti and the BEAST 1.7." Molecular Biology and Evolution 29, no. 8 (August 1, 2012): 1969–73. doi:10.1093/molbev/mss075.

[32] Earl, David J, and Michael W Deem. 2005. "Parallel Tempering: Theory, Applications, and New Perspectives." Physical Chemistry Chemical Physics 7 (23): 3910–16.

[33] Felsenstein, Joseph. Inferring Phylogenies. Vol. 2. Sinauer Associates Sunderland, 2004.

[34] Fischer, Martin C., Matthieu Foll, Laurent Excoffier, and Gerald Heckel. "Enhanced AFLP Genome Scans Detect Local Adaptation in High-Altitude Populations of a Small Rodent (Microtus Arvalis)." Molecular Ecology 20, no. 7 (April 2011): 1450–62. doi:10.1111/j.1365-294X.2011.05015.x.

[35] FitzJohn, Richard G. "Diversitree: Comparative Phylogenetic Analyses of Diversification in R." Methods in Ecology and Evolution 3, no. 6 (December 1, 2012): 1084–92. doi:10.1111/j.2041-210X.2012.00234.x.

[36] Fletcher, William, and Ziheng Yang. 2009. "INDELible: A Flexible Simulator of Biological Sequence Evolution." Molecular Biology and Evolution 26 (8): 1879–88. doi:10.1093/molbev/msp098.

[37] Fredman, M. L., D. S. Johnson, L. A. McGeoch, and G. Ostheimer. "Data Structures for Traveling Salesmen." In Selected Papers from the Fourth Annual ACM SIAM Symposium on Discrete Algorithms, 432–79. SODA '93. Orlando, FL, USA: Academic Press, Inc., 1995. http://dl.acm.org/citation.cfm?id=203612.203628.

[38] Gallager, Robert G. 1962. "Low-Density Parity-Check Codes." Information Theory, IRE Transactions on 8 (1): 21–28.

[39] Gamerman, Dani, and Hedibert F. Lopes. Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference, Second Edition. CRC Press, 2006.

[40] Gelman, A, GO Roberts, and WR Gilks. 1996. "Efficient Metropolis Jumping Rules." In Bayesian Statistics, 5 (Alicante, 1994), 599–607. Oxford Univ. Press.

[41] Geyer, Charles J. "Markov Chain Monte Carlo Maximum Likelihood." Interface Foundation of North America, 1991.

[42] Gilks, W. R., S. Richardson, and David Spiegelhalter. Markov Chain Monte Carlo in Practice. CRC Press, 1995.

[43] Girolami, Mark, and Ben Calderhead. 2011. "Riemann Manifold Langevin and Hamiltonian Monte Carlo Methods." Journal of the Royal Statistical Society: Series B (Statistical Methodology) 73 (2): 123–214. doi:10.1111/j.1467-9868.2010.00765.x.

[44] Goldman, N., and Z. Yang. "A Codon-Based Model of Nucleotide Substitution for Protein-Coding DNA Sequences." Molecular Biology and Evolution 11, no. 5 (September 1994): 725–36.

[45] Gong, Lei, and James M. Flegal. "A Practical Sequential Stopping Rule for High-Dimensional MCMC and Its Application to Spatial-Temporal Bayesian Models." arXiv:1403.5536 [stat], March 21, 2014. http://arxiv.org/abs/1403.5536.

[46] Green, Peter J., and Xiao-liang Han. "Metropolis Methods, Gaussian Proposals and Antithetic Variables." In Stochastic Models, Statistical Methods, and Algorithms in Image Analysis, edited by Piero Barone, Arnoldo Frigessi, and Mauro Piccioni, 142–64. Lecture Notes in Statistics 74. Springer New York, 1992..

[47] Green, Peter J., Krzysztof Łatuszyński, Marcelo Pereyra, and Christian P. Robert. 2015. "Bayesian Computation: A Summary of the Current State, and Samples Backwards and Forwards." Statistics and Computing 25 (4): 835–62. doi:10.1007/s11222-015-9574-5.

[48] Griewank, Andreas, and Andrea Walther. Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Siam, 2008.

[49] Guindon, Stéphane, Allen G. Rodrigo, Kelly A. Dyer, and John P. Huelsenbeck. "Modeling the Site-Specific Variation of Selection Patterns along Lineages." Proceedings of the National Academy of Sciences of the United States of America 101, no. 35 (August 31, 2004): 12957–62. doi:10.1073/pnas.0402177101.

[50] Guindon, Stéphane, Jean-François Dufayard, Vincent Lefort, Maria Anisimova, Wim Hordijk, and Olivier Gascuel. "New Algorithms and Methods to Estimate Maximum-Likelihood Phylogenies: Assessing the Performance of PhyML 3.0." Systematic Biology 59, no. 3 (May 2010): 307–21. doi:10.1093/sysbio/syq010.

[51] Haario, Heikki, Eero Saksman, and Johanna Tamminen. 2001. "An Adaptive Metropolis Algorithm." Bernoulli 7 (2): 223–42.

[52] Haario, Heikki, Eero Saksman, and Johanna Tamminen. 2005. "Componentwise Adaptation for High Dimensional MCMC." Computational Statistics 20 (2): 265–73. doi:10.1007/BF02789703.

[53] Haario, Heikki, Marko Laine, Antonietta Mira, and Eero Saksman. 2006. "DRAM: Efficient Adaptive MCMC." Statistics and Computing 16 (4): 339–54. doi:10.1007/s11222-006-9438-0.

[54] Hagen, Joel B. "The Origins of Bioinformatics." Nature Reviews Genetics 1, no. 3 (December 2000): 231–36. doi:10.1038/35042090.

[55] Hascoet, Laurent, and Ala Taftaf. "On the Correct Application of AD Checkpointing to Adjoint MPI-Parallel Programs." Report. Inria Sophia Antipolis, February 22, 2016. https://hal.inria.fr/hal-01277449/document.

[56] Hastings, W. K. 1970. "Monte Carlo Sampling Methods Using Markov Chains and Their Applications." Biometrika 57 (1): 97–109. doi:10.2307/2334940.

[57] Helsgaun, Keld. "General K-Opt Submoves for the Lin–Kernighan TSP Heuristic." Mathematical Programming Computation 1, no. 2–3 (July 1, 2009): 119–63. doi:10.1007/s12532-009-0004-6.

[58] Hinchliff, Cody E., Stephen A. Smith, James F. Allman, J. Gordon Burleigh, Ruchi Chaudhary, Lyndon M. Coghill, Keith A. Crandall, et al. "Synthesis of Phylogeny and Taxonomy into a Comprehensive Tree of Life." Proceedings of the National Academy of Sciences 112, no. 41 (October 13, 2015): 12764–69. doi:10.1073/pnas.1423041112.

[59] Höhna, Sebastian, Tracy A. Heath, Bastien Boussau, Michael J. Landis, Fredrik Ronquist, and John P. Huelsenbeck. "Probabilistic Graphical Model Representation in Phylogenetics." Systematic Biology 63, no. 5 (September 1, 2014): 753–71. doi:10.1093/sysbio/syu039.

[60] Homan, Matthew D., and Andrew Gelman. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." J. Mach. Learn. Res. 15, no. 1 (January 2014): 1593–1623.

[61] Johnson, David S, and Lyle A McGeoch. "The Traveling Salesman Problem: A Case Study in Local Optimization." Local Search in Combinatorial Optimization 1 (1997): 215–310.

[62] Johnson, David S., and Lyle A. McGeoch. "Experimental Analysis of Heuristics for the STSP." In Local Search in Combinatorial Optimization. Wiley & Sons, 2001.

[63] Johnson, Steven G., "The NLopt nonlinear-optimization package." http://ab-initio.mit.edu/nlopt (January 2016).

[64] Jordan, Michael I. "Graphical Models." Statistical Science 19, no. 1 (February 2004): 140–55. doi:10.1214/088342304000000026.

[65] Jordan, Michal I. "The era of Big Data." The official bulletin of the International Society for Bayesian Analysis 28, no. 2 (June, 2011).

[66] Karypis, George, and Vipin Kumar. 1998. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs." SIAM J. Sci. Comput. 20 (1): 359–92. doi:10.1137/S1064827595287997.

[67] Kasahara, H., and S. Narita. 1984. "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing." IEEE Transactions on Computers C-33 (11): 1023–29. doi:10.1109/TC.1984.1676376.

[68] Koller, Daphne, and Nir Friedman. Probabilistic Graphical Models: Principles and Techniques. MIT press, 2009.

[69] Koziel, Slawomir, Leifur Leifsson, Michael Lees, Valeria V.Krzhizhanovskaya, Jack Dongarra, Peter M.A. Sloot, Markus Towara, Michel Schanen, and Uwe Naumann. "International Conference On Computational Science, ICCS 2015MPI-Parallel Discrete Adjoint OpenFOAM." Procedia Computer Science 51 (January 1, 2015): 19–28.

[70] Kriegel, Hans-Peter, Peer Kröger, and Arthur Zimek. “Clustering High-Dimensional Data: A Survey on Subspace Clustering, Pattern-Based Clustering, and Correlation Clustering.” ACM Trans. Knowl. Discov. Data 3, no. 1 (March 2009): 1:1–1:58. doi:10.1145/1497577.1497578.

[71] Kuhner, Mary K. “LAMARC 2.0: Maximum Likelihood and Bayesian Estimation of Population Parameters.” Bioinformatics 22, no. 6 (March 15, 2006): 768–70. doi:10.1093/bioinformatics/btk051.

[72] Kumar, Suresh Nanda, and Ramasamy Panneerselvam. “A Survey on the Vehicle Routing Problem and Its Variants.” Intelligent Information Management 4, no. 3 (2012): 66.

[73] Kwok, Yu-Kwong, and I. Ahmad. 1996. “Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors.” IEEE Transactions on Parallel and Distributed Systems 7 (5): 506–21. doi:10.1109/71.503776.

[74] Kwok, Yu-Kwong, and Ishfaq Ahmad. 1999. “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors.” ACM Comput. Surv. 31 (4): 406–71. doi:10.1145/344588.344618.

[75] Kwok, Yu-Kwong, and I. Ahmad. 1999. “FASTEST: A Practical Low-Complexity Algorithm for Compile-Time Assignment of Parallel Programs to Multiprocessors.” IEEE Transactions on Parallel and Distributed Systems 10 (2): 147–59. doi:10.1109/71.752781.

[76] Lakner, Clemens, Paul van der Mark, John P. Huelsenbeck, Bret Larget, and Fredrik Ronquist. “Efficiency of Markov Chain Monte Carlo Tree Proposals in Bayesian Phylogenetics.” Systematic Biology 57, no. 1 (February 2008): 86–103. doi:10.1080/10635150801886156.

[77] Lartillot, Nicolas, Nicolas Rodrigue, Daniel Stubbs, and Jacques Richer. “PhyloBayes MPI. Phylogenetic Reconstruction with Infinite Mixtures of Profiles in a Parallel Environment.” Systematic Biology, April 5, 2013, syt022. doi:10.1093/sysbio/syt022.

[78] Lartillot, Nicolas. 2015. “Probabilistic Models of Eukaryotic Evolution: Time for Integration.” Phil. Trans. R. Soc. B 370 (1678): 20140338. doi:10.1098/rstb.2014.0338.

[79] Lepage, Thomas, David Bryant, Hervé Philippe, and Nicolas Lartillot. 2007. “A General Comparison of Relaxed Molecular Clock Models.” Molecular Biology and Evolution 24 (12): 2669–80. doi:10.1093/molbev/msm193.

[80] Lunn, David J, Andrew Thomas, Nicky Best, and David Spiegelhalter. “WinBUGS-a Bayesian Modelling Framework: Concepts, Structure, and Extensibility.” Statistics and Computing 10, no. 4 (2000): 325–37.

[81] Marjoram, Paul, John Molitor, Vincent Plagnol, and Simon Tavaré. 2003. “Markov Chain Monte Carlo without Likelihoods.” Proceedings of the National Academy of Sciences 100 (26): 15324–28. doi:10.1073/pnas.0306899100.

[82] Marx, Vivien. “Biology: The Big Challenges of Big Data.” Nature 498, no. 7453 (June 13, 2013): 255–60. doi:10.1038/498255a.

[83] Mengersen, Kerrie L., Pierre Pudlo, and Christian P. Robert. 2013. “Bayesian Computation via Empirical Likelihood.” Proceedings of the National Academy of Sciences 110 (4): 1321–26. doi:10.1073/pnas.1208827110.

[84] Metropolis, Nicholas, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. “Equation of State Calculations by Fast Computing Machines.” The Journal of Chemical Physics 21 (6): 1087–92. doi:10.1063/1.1699114.

[85] Miasojedow, Błażej, Eric Moulines, and Matti Vihola. 2013. “An Adaptive Parallel Tempering Algorithm.” Journal of Computational and Graphical Statistics 22 (3): 649–64. doi:10.1080/10618600.2013.778779.

[86] Moler, C., and C. Van Loan. “Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later.” SIAM Review 45, no. 1 (January 1, 2003): 3–49. doi:10.1137/S00361445024180.

[87] Moretti, Sébastien, Balazs Laurenczy, Walid H. Gharib, Briséïs Castella, Arnold Kuzniar, Hannes Schabauer, Romain A. Studer, et al. “Selectome Update: Quality Control and Computational Improvements to a Database of Positive Selection.” Nucleic Acids Research 42, no. D1 (January 1, 2014): D917–21. doi:10.1093/nar/gkt1065.

[88] Muse, S. V., and B. S. Gaut. “A Likelihood Approach for Comparing Synonymous and Nonsynonymous Nucleotide Substitution Rates, with Application to the Chloroplast Genome.” Molecular Biology and Evolution 11, no. 5 (September 1, 1994): 715–24.

[89] Naumann, Uwe, Laurent Hascoët, Chris Hill, Paul Hovland, Jan Riehme, and Jean Utke. “A Framework for Proving Correctness of Adjoint Message-Passing Programs.” In Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, 316–321. Berlin, Heidelberg: Springer-Verlag, 2008.

[90] Naumann, Uwe. “The Art of Differentiating Computer Programs.” 2016.

[91] Nocedal, Jorge, and Stephen J. Wright. 2006. Numerical Optimization. 2nd ed. Springer Series in Operations Research. New York: Springer.

[92] Ouzounis, Christos A. “Rise and Demise of Bioinformatics? Promise and Progress.” PLoS Comput Biol 8, no. 4 (April 26, 2012): e1002487. doi:10.1371/journal.pcbi.1002487.

[93] Pattengale, Nicholas D., Masoud Alipour, Olaf R. Bininda-Emonds, Bernard M. Moret, and Alexandros Stamatakis. "How Many Bootstrap Replicates Are Necessary?" In Proceedings of the 13th Annual International Conference on Research in Computational Molecular Biology, 184–200. RECOMB 2'09. Berlin, Heidelberg: Springer-Verlag, 2009. doi:10.1007/978-3-642-02008-7_13.

[94] Marinari, E., and G. Parisi. 1992. "Simulated Tempering: A New Monte Carlo Scheme." EPL (Europhysics Letters) 19 (6): 451. doi:10.1209/0295-5075/19/6/002.

[95] Pearl, Judea. Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988.

[96] Plummer, Martyn. "JAGS: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling," 2003.

[97] Pond, Sergei L. Kosakovsky, Ben Murrell, Mathieu Fourment, Simon D. W. Frost, Wayne Delport, and Konrad Scheffler. "A Random Effects Branch-Site Model for Detecting Episodic Diversifying Selection." Molecular Biology and Evolution 28, no. 11 (November 1, 2011): 3033–43. doi:10.1093/molbev/msr125.

[98] Proux, Estelle, Romain A. Studer, Sébastien Moretti, and Marc Robinson-Rechavi. "Selectome: A Database of Positive Selection." Nucleic Acids Research 37, no. suppl 1 (January 1, 2009): D404–7. doi:10.1093/nar/gkn768.

[99] Pyron, R Alexander, and John J Wiens. 2011. "A Large-Scale Phylogeny of Amphibia Including over 2800 Species, and a Revised Classification of Extant Frogs, Salamanders, and Caecilians." Molecular Phylogenetics and Evolution 61 (2): 543–83.

[100] Rego, César, Dorabela Gamboa, Fred Glover, and Colin Osterman. "Traveling Salesman Problem Heuristics: Leading Methods, Implementations and Latest Advances." European Journal of Operational Research 211, no. 3 (June 16, 2011): 427–41. doi:10.1016/j.ejor.2010.09.010.

[101] Robbins, Herbert, and Sutton Monro. "A Stochastic Approximation Method." The Annals of Mathematical Statistics 22, no. 3 (September 1951): 400–407. doi:10.1214/aoms/1177729586.

[102] Roberts, G. O., A. Gelman, and W. R. Gilks. 1997. "Weak Convergence and Optimal Scaling of Random Walk Metropolis Algorithms." The Annals of Applied Probability 7 (1): 110–20. doi:10.1214/aoap/1034625254.

[103] Roberts, G. O., and S. K. Sahu. "Updating Schemes, Correlation Structure, Blocking and Parameterization for the Gibbs Sampler." Journal of the Royal Statistical Society: Series B (Statistical Methodology) 59, no. 2 (1997): 291–317. doi:10.1111/1467-9868.00070.

[104] Roberts, Gareth O., and Jeffrey S. Rosenthal. "Optimal Scaling for Various Metropolis-Hastings Algorithms." Statistical Science 16, no. 4 (November 2001): 351–67. doi:10.1214/ss/1015346320.

[105] Roberts, Gareth O., and Jeffrey S. Rosenthal. 2009. "Examples of Adaptive MCMC." Journal of Computational and Graphical Statistics 18 (2): 349–67. doi:10.1198/jcgs.2009.06134.

[106] Ronquist, Fredrik, Maxim Teslenko, Paul van der Mark, Daniel L. Ayres, Aaron Darling, Sebastian Hohna, Bret Larget, Liang Liu, Marc A. Suchard, and John P. Huelsenbeck. "MrBayes 3.2: Efficient Bayesian Phylogenetic Inference and Model Choice Across a Large Model Space." Systematic Biology 61, no. 3 (May 2012): 539–42. doi:10.1093/sysbio/sys029.

[107] Sahni, Sartaj K. "Algorithms for Scheduling Independent Tasks." J. ACM 23, no. 1 (January 1976): 116–27. doi:10.1145/321921.321934.

[108] Schabauer, H., M. Valle, C. Pacher, H. Stockinger, A. Stamatakis, M. Robinson-Rechavi, Ziheng Yang, and N. Salamin. "SlimCodeML: An Optimized Version of CodeML for the Branch-Site Model." In Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International, 706–14, 2012. doi:10.1109/IPDPSW.2012.88.

[109] Schaeffer, Satu Elisa. "Graph Clustering." Computer Science Review 1, no. 1 (August 2007): 27–64. doi:10.1016/j.cosrev.2007.05.001.

[110] Schanen, Michel, Uwe Naumann, Laurent Hascoët, and Jean Utke. "ICCS 2010 Interpretative Adjoints for Numerical Simulation Codes Using MPI." Procedia Computer Science 1, no. 1 (2010): 1825–33.

[111] Scott, Steven L., Alexander W. Blocker, and Fernando V. Bonassi. 2013. "Bayes and Big Data: The Consensus Monte Carlo Algorithm." In Bayes 250.

[112] Schanen, Michel, Uwe Naumann, Laurent Hascoët, and Jean Utke. "ICCS 2010 Interpretative Adjoints for Numerical Simulation Codes Using MPI." Procedia Computer Science 1, no. 1 (2010): 1825–33. doi:http://dx.doi.org/10.1016/j.procs.2010.04.204.

[113] Sherlock, Chris, Andrew Golightly, and Daniel A. Henderson. 2015. "Adaptive, Delayed-Acceptance MCMC for Targets with Expensive Likelihoods." arXiv:1509.00172 [math, Stat], September. http://arxiv.org/abs/1509.00172.

[114] Silvestro, Daniele, Jan Schnitzler, Lee Hsiang Liow, Alexandre Antonelli, and Nicolas Salamin. "Bayesian Estimation of Speciation and Extinction from Incomplete Fossil Occurrence Data." Systematic Biology, February 8, 2014, syu006. doi:10.1093/sysbio/syu006.

[115] Silvestro, Daniele, Nicolas Salamin, and Jan Schnitzler. “PyRate: A New Program to Estimate Speciation and Extinction Rates from Incomplete Fossil Data.” Methods in Ecology and Evolution 5, no. 10 (October 1, 2014): 1126–31. doi:10.1111/2041-210X.12263.

[116] Silvestro, Daniele, Borja Cascales-Miñana, Christine D. Bacon, and Alexandre Antonelli. “Revisiting the Origin and Diversification of Vascular Plants through a Comprehensive Bayesian Analysis of the Fossil Record.” New Phytologist, January 1, 2015, n/a – n/a. doi:10.1111/nph.13247.

[117] Sisson, Scott A. 2005. “Transdimensional Markov Chains: A Decade of Progress and Future Perspectives.” Journal of the American Statistical Association 100: 1077–89.

[118] Sokal, Robert R. “A Statistical Method for Evaluating Systematic Relationships.” Univ Kans Sci Bull 38 (1958): 1409–38.

[119] Srivastava, Sanvesh, Cheng Li, and David B. Dunson. 2015. “Scalable Bayes via Barycenter in Wasserstein Space.” arXiv:1508.05880 [stat], August. http://arxiv.org/abs/1508.05880.

[120] Stamatakis, Alexandros. “RAxML Version 8: A Tool for Phylogenetic Analysis and Post-Analysis of Large Phylogenies.” Bioinformatics, January 21, 2014, btu033. doi:10.1093/bioinformatics/btu033.

[121] Stan Development Team. “Stan, Version 2.9.0” (2016)

[122] Stephens, Zachary D., Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. “Big Data: Astronomical or Genomical?” PLoS Biol 13, no. 7 (July 7, 2015): e1002195. doi:10.1371/journal.pbio.1002195.

[123] Strid, Ingvar. “Efficient Parallelisation of Metropolis–Hastings Algorithms Using a Prefetching Approach.” Computational Statistics & Data Analysis 54, no. 11 (November 1, 2010): 2814–35. doi:10.1016/j.csda.2009.11.019.

[124] Tavaré, Simon. “Some Probabilistic and Statistical Problems in the Analysis of DNA Sequences.” Lectures on Mathematics in the Life Sciences 17 (1986): 57–86.

[125] Tierney, Luke. “Markov Chains for Exploring Posterior Distributions.” The Annals of Statistics 22, no. 4 (December 1994): 1701–28. doi:10.1214/aos/1176325750.

[126] Utke, J., L. Hascoet, P. Heimbach, C. Hill, P. Hovland, and U. Naumann. “Toward Adjoinable MPI.” In IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009, 1–8, 2009. doi:10.1109/IPDPS.2009.5161165.

[127] Valle, Mario, Hannes Schabauer, Christoph Pacher, Heinz Stockinger, Alexandros Stamatakis, Marc Robinson-Rechavi, and Nicolas Salamin. “Optimization Strategies for Fast Detection of Positive Selection on Phylogenetic Trees.” Bioinformatics, January 2, 2014, btt760. doi:10.1093/bioinformatics/btt760.

[128] Vihola, Matti. 2012. “Robust Adaptive Metropolis Algorithm with Coerced Acceptance Rate.” Statistics and Computing 22 (5): 997–1008. doi:10.1007/s11222-011-9269-5.

[129] Volin, Yu. M., and G. M. Ostrovskii. “Automatic Computation of Derivatives with the Use of the Multilevel Differentiating technique—1. Algorithmic Basis.” Computers & Mathematics with Applications 11, no. 11 (November 1, 1985): 1099–1114.

[130] Vitti, Joseph J., Sharon R. Grossman, and Pardis C. Sabeti. “Detecting Natural Selection in Genomic Data.” Annual Review of Genetics 47, no. 1 (2013): 97–120. doi:10.1146/annurev-genet-111212-133526.

[131] Wainwright, Martin J., and Michael I. Jordan. “Graphical Models, Exponential Families, and Variational Inference.” Foundations and Trends® in Machine Learning 1, no. 1–2 (2007): 1–305. doi:10.1561/2200000001.

[132] Wang, Xiangyu, Fangjian Guo, Katherine A Heller, and David B Dunson. 2015. “Parallelizing MCMC with Random Partition Trees.” In Advances in Neural Information Processing Systems 28, edited by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, 451–59. Curran Associates, Inc.

[133] Willebeek-LeMair, M. H., and A. P. Reeves. “Strategies for Dynamic Load Balancing on Highly Parallel Computers.” IEEE Trans. Parallel Distrib. Syst. 4, no. 9 (September 1993): 979–93. doi:10.1109/71.243526.

[134] Yip, Kevin Y., Prianka Patel, Philip M. Kim, Donald M. Engelman, Drew McDermott, and Mark Gerstein. “An Integrated System for Studying Residue Coevolution in Proteins.” Bioinformatics 24, no. 2 (January 15, 2008): 290–92. doi:10.1093/bioinformatics/btm584.

[135] Yang, Ziheng. “PAML: Phylogenetic Analysis by Maximum Likelihood.” Molecular Biology and Evolution 24 (1993): 1586–91.

[136] Yang, Ziheng, and Joseph P. Bielawski. “Statistical Methods for Detecting Molecular Adaptation.” Trends in Ecology & Evolution 15, no. 12 (December 1, 2000): 496–503. doi:10.1016/S0169-5347(00)01994-7.

[137] Yang, Ziheng, Rasmus Nielsen, Nick Goldman, and Anne-Mette Krabbe Pedersen. “Codon-Substitution Models for Heterogeneous Selection Pressure at Amino Acid Sites.” Genetics 155, no. 1 (May 1, 2000): 431–49.

[138] Yang, Ziheng, and Rasmus Nielsen. “Codon-Substitution Models for Detecting Molecular Adaptation at Individual Sites Along Specific Lineages.” Molecular Biology and Evolution 19, no. 6 (June 1, 2002): 908–17.

[139] Yang, Ziheng. 2006. Computational Molecular Evolution. Vol. 284. Oxford University Press Oxford.

[140] Yang, Ziheng. "PAML 4: Phylogenetic Analysis by Maximum Likelihood." Molecular Biology and Evolution 24, no. 8 (2007): 1586–91.

Appendices

A. Algorithms

A.1. Dynamic load balancing

Algorithm 13 Dynamic load balancing: evaluate likelihood - processor p

```
// Global node buffer C = ∅, traversed = false
// Init thread local node buffer with ready nodes
B_p = init(p)
// getNextNode(·) is defined in Algo. 14
while (v_p = getNextNode(B_p)) ≠ ∅ do
  process(v)
  if ν(v_p) ≠ ∅ then
    // updateDAG(·) is defined in Algo. 15
    updateDAG(v_p, B_p)
  else
    traversed = true
    signalAll(EmptyNodes)
  end if
end while
```

Algorithm 14 Dynamic load balancing: get next node - processor p

```
v_p = ∅
if B_p ≠ ∅ then
  v_p = pop(B_p)
else
  lock(GlobalLock)
  while C = ∅ ∧ ¬traversed do
    wait(EmptyNodes)
  end while
  if C ≠ ∅ then
    v_p = pop(C)
  end if
  unlock(GlobalLock)
end if
return v_p
```

Algorithm 15 Dynamic load balancing: update DAG - processor p

```
𝓑_tmp = ∅
// Signal that dependencies is fulfilled to parents
for v_i ∈ ν(v_p) do
  lock(getLockFromPool(v_i))
  signalProcessed(v_i, v_p)
  if ready(v_i) then
    if empty(𝓑_p) then
      // Add first ready parent to local buffer
      𝓑_p = v_i
    else
      // Add other ready parent to temp.  buffer
      𝓑_tmp = 𝓑_tmp ∪ v_i
    end if
  end if
  unlock(getLockFromPool(v_i))
end for

// Add ready nodes to 𝓒
lock(GlobalLock)
for v_i ∈ 𝓑_tmp do
  𝓒 = 𝓑 ∪ v_i
  signal(EmptyNodes)
end for
unlock(GlobalLock)
```

B. Theorems

B.1. Proof of strict triangle inequality for the scheduling of gradient approximations

Some definitions based on the content of chapters 2 and 3 are introduced prior to the theorem and its proof.

Definition 1. A change in value of a parameter θ_i always induces computations, therefore

$$\forall i \in (1..m)\,,\ \Psi_{\theta_i} \neq \emptyset$$

The notation Ψ_{θ_i} is equivalent to $\Psi_{\{\theta_i\}}$.

Definition 2. The set of vertices $\Psi_{\mathcal{A}}$ is equivalent to the union of the set of vertices induced by change of single parameter such that

$$\Psi_{\mathcal{A}} = \bigcup\nolimits_{i \in \mathcal{A}} \Psi_{\theta_i}$$

Definition 3. The cost function $\mathcal{C}$ acts as a form of weighted count of the set $\Psi_{\mathcal{A}}$ in function of vertex weights $\chi_v > 0 : v \in \Psi_{\mathcal{A}}$, as

$$\mathcal{C}\left(\Psi_{\mathcal{A}}\right) = \sum_{v \in \Psi_{\mathcal{A}}} \chi_v \ \rightarrow\ \mathcal{C}\left(\Psi_{\mathcal{A}}\right) > 0.$$

By extension this *counting* function follows the *principle of inclusion-exclusion*, implying that for the cost of a set $\mathcal{A} = \{a, b\}$ can be decomposed as

$$\mathcal{C}\left(\Psi_{\mathcal{A}}\right) = \mathcal{C}\left(\Psi_{\{a,b\}}\right) = \mathcal{C}\left(\Psi_a \cup \Psi_b\right) = \mathcal{C}\left(\Psi_a\right) + \mathcal{C}\left(\Psi_b\right) - \mathcal{C}\left(\Psi_a \cap \Psi_b\right).$$

Definition 4. Given that $\left(\Psi_a \cap \Psi_b\right) \subseteq \Psi_b$, the cost $\mathcal{C}$ of the intersection of two sets cannot exceed the cost of one of the sets, such that

$$\mathcal{C}\left(\Psi_a \cap \Psi_b\right) \leq \mathcal{C}\left(\Psi_b\right).$$

Theorem 5. *The distance, or cost $\mathcal{C}$, between a pair of perturbations used to define the instance of the travelling salesman problem (Eq.* (3.10) *and* (3.11)*) is subject to the strict triangle inequality defining that*

$$\mathcal{C}\left(\Psi_{\{\theta_a,\theta_c\}}\right) < \mathcal{C}\left(\Psi_{\{\theta_a,\theta_b\}}\right) + \mathcal{C}\left(\Psi_{\{\theta_b,\theta_c\}}\right)$$

with a, b, c representing the index $i \in (1..m)$ of the perturbations $e_i\epsilon$ and Ψ_{θ_i} representing the set of vertices in the directed task graph that must be recomputed following a perturbation of parameter θ_i.

Proof. Starting from the strict triangle inequality defined as

$$\mathcal{C}\left(\Psi_{\{\theta_a,\theta_c\}}\right) < \mathcal{C}\left(\Psi_{\{\theta_a,\theta_b\}}\right) + \mathcal{C}\left(\Psi_{\{\theta_b,\theta_c\}}\right).$$

The cost function can be decomposed in function of the inclusion-exclusion principle as,

$$\begin{aligned}
\mathcal{C}(\Psi_a) + \mathcal{C}(\Psi_c) - \mathcal{C}(\Psi_a \cap \Psi_c) < \mathcal{C}(\Psi_a) + \mathcal{C}(\Psi_b) - \mathcal{C}(\Psi_a \cap \Psi_b) \\
+\mathcal{C}(\Psi_b) + \mathcal{C}(\Psi_c) - \mathcal{C}(\Psi_b \cap \Psi_c), \quad \text{(B.1)} \\
-\mathcal{C}(\Psi_a \cap \Psi_c) < 2 \times \mathcal{C}(\Psi_b) - \left[\mathcal{C}(\Psi_a \cap \Psi_b) + \mathcal{C}(\Psi_b \cap \Psi_c)\right], \quad \text{(B.2)} \\
\mathcal{C}(\Psi_a \cap \Psi_b) + \mathcal{C}(\Psi_b \cap \Psi_c) < 2 \times \mathcal{C}(\Psi_b) + \mathcal{C}(\Psi_a \cap \Psi_c). \quad \text{(B.3)}
\end{aligned}$$

Given that $\mathcal{C}(\cdot) \geq 0$ and $\mathcal{C}(\Psi_b) > 0$ (Def. 1), the left hand side must at least be equal to $2 \times \mathcal{C}(\Psi_b)$ to prove that the strict triangular inequality does not hold. Furthermore, the costs related to the set intersection $(\Psi_a \cap \Psi_b)$ or $(\Psi_b \cap \Psi_c)$ cannot exceed this cost $\mathcal{C}(\Psi_b)$ as defined in condition 4. Therefore two cases are identified.

Case 1. At least one of the intersections $(\Psi_a \cap \Psi_b)$ or $(\Psi_b \cap \Psi_c)$ is strictly included in Ψ_b, such that

$$\begin{aligned}
& \mathcal{C}(\Psi_a \cap \Psi_b) < \mathcal{C}(\Psi_b) \text{ or } \mathcal{C}(\Psi_b \cap \Psi_c) < \mathcal{C}(\Psi_b) \\
\Downarrow \\
& \mathcal{C}(\Psi_a \cap \Psi_b) + \mathcal{C}(\Psi_b \cap \Psi_c) < 2 \times \mathcal{C}(\Psi_b).
\end{aligned}$$

Therefore equation B.3 holds and the strict triangle inequality is proven even if

$$\mathcal{C}(\Psi_a \cap \Psi_c) = 0.$$

Case 2. Both sets of intersections $(\Psi_a \cap \Psi_b)$ and $(\Psi_b \cap \Psi_c)$ are equal to Ψ_b, such that $\mathcal{C}(\Psi_a \cap \Psi_b) = \mathcal{C}(\Psi_b \cap \Psi_c) = \mathcal{C}(\Psi_b)$. Therefore equation B.3 can be reformulated as

$$\begin{aligned}
2 \times \mathcal{C}(\Psi_b) < 2 \times \mathcal{C}(\Psi_b) + \mathcal{C}(\Psi_a \cap \Psi_c), \\
0 < \mathcal{C}(\Psi_a \cap \Psi_c).
\end{aligned}$$

Given that $\Psi_b \subseteq \Psi_a$ and $\Psi_b \subseteq \Psi_c$, then $\Psi_b \subseteq (\Psi_a \cap \Psi_c)$ such that

$$0 < \mathcal{C}(\Psi_b) \leq \mathcal{C}(\Psi_a \cap \Psi_c).$$

Therefore, in this second case the strict triangular inequality also holds.

□

C. Detailed MCMC experiments

All presented analysis were done on a dedicated cluster partition of 8 Intel Xeon X5650 (2.67GHz) nodes with 64GB of RAM at the University of Geneva. `MrBayes` measures were done with the version 3.2.5 compiled with SSE support and MPI enabled.

C.1. Codon-substitution models

C.1.1. Settings

Two different evolutionary trees were simulated using `INDELible` : the first one `Dataset 1` had 16 taxa and the second one `Dataset 2` had 32 taxa. For each evolutionary tree, two separate alignments of 100 codons were simulated under mild purifying selection ($\omega = 0.8$).

Each method was run 4 times per simulated alignments with different starting random seed for $200 \cdot 10^3$ iterations. Therefore each dataset consisted of 8 independent MCMC runs. Measures were applied on the obtained samples after removing a burn-in phase of $40 \cdot 10^3$ samples (20%). The presented ESS and run time on each dataset correspond to the average measures over all respective dataset runs.

C.1.2. Measures

Detailed measures for these experiments are shown in table C.1 and C.2

Table C.1.: Detailed results for `Dataset 1`

`MrBayes`					
Nb. Proc.	1	4	8	16	32
Avg. ESS	446.69	-	-	-	-
Avg. Time [s]	2189.93	-	-	-	-
`Default 1PPB`					
Nb. Proc.	1	4	8	16	32
Avg. ESS	692.82	1010.42	913.94	900.73	801.78
Avg. Time [s]	971.72	891.84	839.13	795.98	859.40
`PCA 12PPB`					
Nb. Proc.	1	4	8	16	32
Avg. ESS	1006.67	3329.60	4972.90	6446.45	7574.92
Avg. Time [s]	2008.97	1967.85	1871.64	1950.95	2009.62

Table C.2.: Detailed results for `Dataset 2`

`MrBayes`					
Nb. Proc.	1	4	8	16	32
Avg. ESS	212.84	-	-	-	-
Avg. Time [s]	3676.83	-	-	-	-
`Default 1PPB`					
Nb. Proc.	1	4	8	16	32
Avg. ESS	356.85	479.40	432.09	371.37	334.15
Avg. Time [s]	1196.94	1090.12	980.53	915.22	980.07
`PCA 12PPB`					
Nb. Proc.	1	4	8	16	32
Avg. ESS	556.51	1860.64	2743.32	3494.99	4021.09
Avg. Time [s]	2556.18	2405.78	2274.70	2191.97	2393.21

C.2. Convergence on phylogeny inference

C.2.1. Settings

Twenty runs with different seeds were simulated for each different methods for $\approx 6 \cdot 10^6$ iterations for `M2017` and $\approx 4 \cdot 10^6$ iterations for `M2152`. These runs started from the same initial random tree (generated by MrBayes). For each method, we found sufficient to drop the top three worst runs in order to remove run failing to reach convergence.

The average standard deviation of split frequency was measured according to the following protocol :

1. we ensured that the split frequencies were the same for long MCMC run of `MrBayes` and our own implementation:
2. these reference split frequencies were then used to define the pairwise ASDSF of each run of the experiment.

The pairwise ASDSF was computed using tree log files resulting from `MrBayes` and our implementation by a tool made for the occasion in order to ensure comparable results.

C.2.2. Measures

Detailed measures for these experiments are shown in figure C.1 for empirical dataset `M2017` and figure C.3 for empirical dataset `M2152`. Each figure has four plots representing in descending order :

1. the number of **sample** required to reach the threshold of 0.05 ASDSF;
2. the number of **samples** per **iterations** (gain from prefetching);

3. the time required to compute one **iteration** (total number of iterations divided by total run time);

4. the time required to reach threshold of 0.05 ASDSF.

The red squares represent the average while red lines represent the median.

Figure C.1.: Detailed measures for empiric dataset M2017

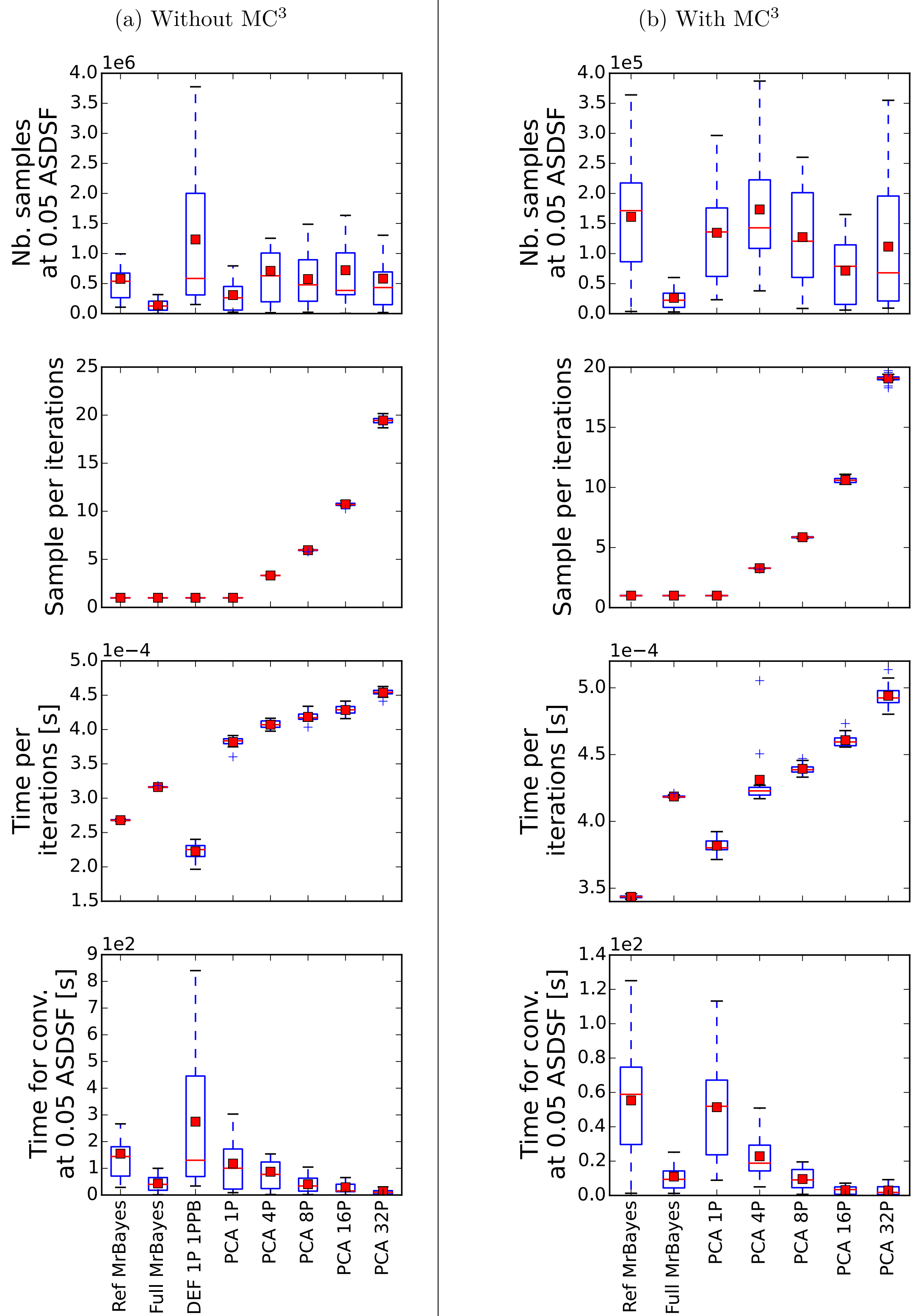


Figure C.3.: Detailed measures for empiric dataset M2152