Inference of Many-Taxon Phylogenies

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Fernando Izquierdo-Carrasco

aus Soria, Spanien

Tag der mündlichen Prüfung:	03.06.2014
Erster Gutachter:	Prof. Dr. Alexandros Stamatakis
Zweiter Gutachter:	Prof. Dr. Arndt von Haeseler

Hiermit erkläre ich, dass ich diese Arbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe.
Ich habe die Satzung der Universität Karlsruhe (TH) zur Sicherung guter wissenschaftlicher Praxis beachtet.

Heidelberg, 30. 01. 2014 ________________________
Fernando Izquierdo-Carrasco

Zusammenfassung

Phylogenetische Bäume stellen evolutionären Beziehungen zwischen verschiedenen Organismen (Arten) dar, die vermutlich gemeinsame Vorfahren haben. Äußere Knoten eines phylogenetischen Baumes (taxa) bezeichnen lebendige Arten, für die molekulare Daten verfügbar sind oder sequenziert werden können. Innere Knoten bezeichnen hypothetische ausgestorbene Arten, für die in der Regel keine Datenquelle zur Verfügung steht.

In den letzten Jahren hat sich das Feld durch die Einführung von sogenannten NGS (Next-Generation-Sequencing) Methoden dramatisch geändert. Diese Methoden, die sich rasch etablierten, erlauben einen erhöhten Durchsatz. Aus diesem Grund wachsen aktuelle molekulare Datenbanken schneller als vom mooreschen Gesetz vorhergesagt. Dadurch entsteht die Herausforderung, solche große molekulare Datenmengen effizient zu analysieren.

Die Maximum-Likelihood Baumrekonstruktion versucht, den Baum mit dem höchsten Likelihood score für ein bestimmtes Sequenzalignment (input) zu finden. Von Seiten der Informatik besteht die Herausforderung darin, die Phylogenetic Likelihood Function (PLF) zum Bewerten alternativer Topologien effizient zu berechnen.

In dieser Arbeit beschäftigen wir uns mit der Untersuchung und Entwicklung von Methoden, die diese Aufgabe, im Zusammenhang mit groß angelegten Datensätzen, erleichtern können. Wir präsentieren drei unterschiedliche Ansätze hierfür: den Speicherbedarf der PLF Funktion zu verringern, Laufzeiten zu reduzieren und die Automatisierung phylogenetischer Analysen.

Verringerung des Speicherbedarfs der PLF Funktion Wir entwickelten drei Methoden, die den Speicherbedarf für Zwischenergebnisse verringern kann. Die (i) *out-of-core* und die (ii) *recomputation* haben miteinander gemeinsam, dass nur ein Teil der Zwischenergebnisse im Hauptspeicher gespeichert werden muss. Die restlichen Zwischenergebnisse werden entweder auf der Festplatte (*out-of-core*) gespeichert oder erneut berechnet (*recomputation* trade-off).

Unsere Auswertungen deuten darauf hin, dass der *recomputation* Ansatz (trade-off) deutlich effizienter ist, da er für typische Datensätze den

Speicherbedarf halbiert zu Lasten einer um 10% erhöhten Laufzeit.

Die dritte Methode wird SEV (Subtree Equality Vectors) genannt und reduziert den Speicherbedarf nahezu proportional zum Anteil fehlender Daten im Datensatz ohne die Laufzeit zu erhöhen. Im allgemeinen können diese Methoden gleichzeitig eingesetzt werden.

Reduzierung von Laufzeiten Wir portierten die PLF Implementierung auf GPUs mit OpenCL. Dabei passten wir das Speicherlayout an, um eine optimale GPU Leistung zu erreichen. Unsere Auswertungen zeigen, dass für ausreichend lange Datensätze, die Verlagerung von Berechnungen auf die GPU bis zu zweimal schneller als der am effizientesten vektorisierte CPU-Code sein kann.

Aus einer algorithmischen Perspektive haben wir den sogenannten *Backbone Algorithm*, der den Suchraum aller möglichen Bäume beschränkten kann, entwickelt. Die Anwendung dieser Methode kann die Laufzeit bis zu 50% verringern und liefert dabei Bäume, die statistisch vergleichbar sind mit denen, die ohne Suchraumbeschränkungen gefunden werden.

Automatisierung phylogenetischer Analysen Der letzte Teil dieser Arbeit beschäftigt sich mit dem Problem, dass vorhandene phylogenetische Bäume nach kurzer Zeit nicht mehr die aktuellsten Daten aus den schnell wachsenden Datenbanken widerspiegeln. Wir entwickelten ein Framework namens *PUmPER* (Phylogenies Updated PERpertually). Mit *PUmPER* können Pipelines erstellt werden, die iterativ neue Sequenzalignments assemblieren und Bäume aus vorherigen Iterationen ergänzen. Das Framework kann entweder als stand-alone pipeline konfiguriert werden oder rechenintensive Aufgaben auf einem Cluster ausführen.

Obwohl die in dieser Arbeit beschriebene Methoden als proof-of-concept innerhalb der RAxML codebase implementiert wurden, sind diese Methoden nichtsdestotrotz relativ einfach in anderen state-of-the-art Likelihood-basierten Codes implementierbar. Im allgemeinen erwarten wir, dass diese Methoden hilfreich sind, um aktuelle phylogenetische Anwendungen besser zu skalieren und dadurch auch phylogenetische Analysen ermöglichen, die auf kompletten Genomen basieren.

Acknowledgements

This research project would not have been possible without the support of many people. I wish to express my gratitude to my supervisor, Prof. Dr. Alexandros Stamatakis who was abundantly helpful and offered invaluable assistance, support and dedicated guidance. Deepest gratitude also to Prof. Dr. Arndt von Haeseler without whose knowledge and assistance this study would not have been successful. I addition I would like to express my sincere gratitude to Casey Dunn, Stephen A. Smith, and John Cazes for kindly hosting me at their lab during my research stay at Brown University. Special thanks also to all my collaborators and fellow group members; Nikos Alachiotis, Simon Berger, Andre Aberer, Pavlos Pavlidis, Solon P. Pissis, Tomas Flouri, Dilrini de Silva, Paschalia Kapli, Kassian Kobert, Jiajie Zhang and Alexey Kozlov for being an always-collaborative and helpful group.

I would like to thank my parents, brothers and family for their unconditional support, as well as my friends from Valladolid and Deggendorf, who have always stood by me during my time in Heidelberg. Last but not least, my dearest thanks to Beifei for being an endless source of encouragement and inspiration.

This work was funded via the German Science Foundation (DFG) grants STA 860/2 and STA 860/3. Part of this work used resources provided by the iPlant Collaborative (funded by NSF grant #DBI-0735191).

Contents

Zusammenfassung **i**

Acknowledgements **iii**

1 Introduction **1**
1.1 Motivation 1
1.2 Scientific Contribution 2
1.3 Structure of this thesis 4

2 Computational Molecular Phylogenetics **5**
2.1 Statistical Models of Evolution 5
2.2 Sequence Alignment 7
2.2.1 Pairwise Sequence Alignment 8
2.2.2 Multiple Sequence Alignment 9
2.3 Phylogenetic Trees 11
2.4 Tree Reconstruction Methods 12
2.5 Computing the Likelihood of a Tree 15
2.5.1 Accounting for rate heterogeneity 19
2.6 Maximum Likelihood Tree Search 21
2.7 Phylogenetic Likelihood Library 24
2.8 RAxML-family implementation concepts 25
2.8.1 Node records 25
2.8.2 Internal tree nodes 25
2.8.3 Data structure for Trees 26
2.8.4 Computing the likelihood on a Tree 27
2.8.5 Optimizing branch lengths 29

3 Memory-Saving Techniques **31**
3.1 Memory requirements for the PLF 32
3.2 Out-of-Core 33
3.2.1 Related Work 34

3.2.2 Computing the PLF Out-of-Core 34
3.2.3 Experimental Setup & Results 39
3.3 Recomputation of Ancestral Vectors 44
3.3.1 Recomputation of Ancestral Probability Vectors 45
3.3.2 Experimental Setup and Results 54
3.3.3 Evaluation of traversal overhead 56
3.4 Subtree Equality Vectors 58
3.4.1 Gappy Subtree Equality Vectors 58
3.4.2 Generation of Biological Test Datasets 61
3.4.3 SEV Performance 62
3.5 Summary 63

4 The Backbone Algorithm 65
4.1 Constraining tree search to a backbone tree 65
4.2 Algorithm 66
4.2.1 Starting tree 67
4.2.2 Tip Clustering 68
4.2.3 Backbone construction 71
4.2.4 Backbone-constrained Tree Search 73
4.3 Evaluation and Results 74
4.3.1 Performance 74
4.3.2 Simulated Datasets (Accuracy) 78
4.4 Summary 78

5 Introduction to GPU Programming 79
5.1 Overview 79
5.2 CUDA 80
5.2.1 CUDA Hardware and Architecture 80
5.2.2 CUDA Programming Model 82
5.2.3 Performance Considerations 82
5.3 OpenCL 84
5.3.1 OpenCL performance portability 85

6 GPU implementation of Phylogenetic Kernels 86
6.1 Related work 87
6.2 Generic Vectorization 88
6.3 GPU Implementation 91
6.3.1 Kernel Implementation 92
6.3.2 GPU Memory Organization 94
6.3.3 OpenCL Implementation 95
6.4 Experimental setup and results 96

6.5 Summary 100

7 Perpetual Phylogenies with PUmPER 101

7.1 Related Work 102

7.2 Framework Overview 103

7.2.1 MSA Construction/Extension with PHLAWD 105

7.2.2 Phylogenetic Inference 107

7.2.3 Manual and automatic tree updates 108

7.3 Software and Availability 108

7.3.1 Standalone implementation 110

7.3.2 Distributed implementation 110

7.3.3 Custom iPlant setup 111

7.4 Evaluation and Results 113

7.4.1 Biological examples 113

7.4.2 Simulated Data 119

7.5 Discussion 121

7.6 Summary 121

8 Conclusion And Outlook 122

8.1 Conclusion 122

8.2 Future Work 123

8.2.1 GPUs 123

8.2.2 PUmPER 124

8.3 Outlook 124

List of Figures 126

List of Tables 128

List of Acronyms 129

Bibliography 130

Chapter 1

Introduction

1.1 Motivation

The landscape of genomics and phylogenetics has changed dramatically since the irruption of Next-Generation Sequencing (NGS) technologies [75]. As a consequence of the throughput increases, the amount of data accumulated in molecular databases keeps growing at an accelerated pace. A current overview over NGS technology platforms is available in [51]. This data avalanche facilitates the exploration of new research areas, such as phylogenomics, where phylogenetic techniques are applied to infer evolutionary relationships based on multi-species genomes [63]. Furthermore, the reduced sequencing cost has enabled the execution of collaborative large-scale genome sequencing projects, such as the 10K vertebrate genome project (`http://www.genome10k.org/`) and the 1000 insect transcriptome evolution project (`http://www.1kite.org/`). The size of the datasets used in these projects pose new challenges for large-scale maximum likelihood-based phylogenetic analyses.

On the other hand, computing power has been growing exponentially in the last decades as predicted by Moore's law. However, at present, due to the high throughput of new sequencing technologies, molecular sequence data accumulates at a pace faster than Moore's law. This induces a growing gap between the data that is available and the computing resources that can be used for analyzing these data.

Phylogenetic trees are tree topologies that represent the evolutionary history of a set of organisms. The field of computational phylogenetics entails the inference of phylogenetic trees. The input data for phylogenetic inference are generally a pre-processed (aligned) set of molecular sequences. The main computational challenge consists in efficiently computing the PLF (Phyloge-

netic Likelihood Function) for scoring alternative tree topologies.

Given the above considerations, in this thesis, we identified and addressed the following computational challenges in the field of computational phylogenetics. Firstly, we aim to reduce the high memory requirements of the PLF to allow for analyzing larger datasets with the available hardware. Secondly, we intend to decrease the running time of tree inferences, either by algorithmic means or architecture-specific optimizations, for instance, using graphic cards (GPUs). Finally, we also study and provide solutions related to the automation of phylogenetic analysis (sequence alignment generation and tree inference). In this case, the focus is not on computational optimizations to reduce memory footprints or achieving speedups, but rather man-hours spent in planning and running phylogenetic analyses. The automation of computational workflows will become an increasingly important task in genomics as manual inspection and analysis of datasets becomes unpractical at the genome scale.

1.2 Scientific Contribution

In this thesis, we explored several areas relevant to the inference of phylogenetic trees under maximum likelihood. The phylogenetic likelihood function (PLF) introduced by Joe Felsenstein in 1981 [23] is one of the most important statistical functions in the area of evolutionary Bioinformatics.

The existing open source software RAxML [87], is a widely used tool for phylogenetic inference. In RAxML the PLF dominates inference times (typically accounting for 80 - 95% of total execution time) and overall memory requirements (accounting for at least 70% of total RAM consumption).

In this thesis, we used RAxML as a platform to implement and test new search algorithms, port the PLF to GPUs, implement memory saving techniques, and to infer extremely large plant trees on real biological data. These *ad hoc* implementations are freely available as open-source code under the GNU GPL license. While we have used RAxML as a workbench, the techniques described in this dissertation are conceptually generic enough to be applied to other likelihood-based state-of-the-art programs for phylogenetic inference such as IQPNNI [53], GARLI [116], PHYML 3.0 [29], FastTree 2.0 [66], MrBayes [72], PhyloBayes [46], and BEAST [17], as well as for PLF libraries such as BEAGLE [4], and other phylogenetic applications such as DPPDiv [31], which requires computing the PLF to estimate divergence times on fixed topologies.

We have developed the *backbone algorithm*, which can reduce the time required for tree inferences by more than 50% while yielding 'good' trees in

the statistical sense. This is achieved by constraining the space of possible topologies that are evaluated by the PLF.

Regarding memory requirements, we present three different techniques that can be applied to compute the likelihood score for a given tree while reducing memory usage. The first two techniques, named out-of-core and recomputation approaches, reduce memory requirements at the cost of longer running times. The out-of-core approach stores on disk the intermediate results that do not fit into memory. The recomputation approach is a trade off where additional computations are used to re-compute intermediate results when not enough RAM memory is available to store them.

The third memory-saving technique presented, based on the SEV (Subtree Equality Vectors) technique, implements the PLF so that some computations are skipped when data is not present. This can reduce memory requirements almost proportionally to the amount of missing data, and, in some cases, it can also significantly reduce running time. However, it is only applicable to input datasets which show certain patterns of missing data.

BEAGLE [4] is a general purpose library for evaluating the likelihood of sequence evolution on trees. One major advantage of using such software libraries is that they often give the user access to parallelized and optimized function kernel implementations. The exilixis-lab (`http://exelixis-lab.org/`) is currently developing the Phylogenetic Likelihood Library (PLL) [27]. Based on the optimized and parallelized PLF implementation of RAxML, the PLL is a highly optimized open-source library that entails state-of-the art implementations for common input datatypes (currently DNA and protein data). The PLL can be executed transparently on a large number of emerging parallel architectures. In the context of this project, we developed a proof-of-concept OpenCL GPU implementation of the key PLF functions for DNA data.

One of the consequences of the rapid growth of molecular databases is that existing phylogenies, after short periods of time, may not reflect the latest available data. Furthermore, re-starting phylogeny reconstruction from scratch to add new data is an expensive task that involves computational resources and man-hours. In order to address this issue, we have developed `PUmPER` (Phylogenies Updated PERpetually), a framework for developing automated pipelines for phylogenetic analysis. `PUmPER` can automatically update existing phylogenies by iteratively assembling new alignments and extending existing trees, without human intervention. The framework can be configured to run as a stand-alone pipeline on a single machine, or to offload computationally expensive tasks to a cluster. `PUmPER` is written in Ruby and transparently uses state-of-the-art tools for alignment construction (PHLAWD [81]) and phylogenetic inference (RAxML-Light [91]).

The main results presented in this thesis have been published in four peer-reviewed conferences [35, 37, 39, 40] and two journal articles [36, 38].

During the course of these thesis, we collaborated and conducted additional work on other research topics not presented here. These projects involved the development and usage of the PLL library [15, 27], heuristic algorithms for the protein assignment problem [30], integration of the recomputation technique in RAxML-Light [91], and conducting phylogenetic analysis in the context of metagenomic species profiling [101]. We also collaborated with the on-going 1000 insect transcriptome evolution project (`http://www.1kite.org/`).

1.3 Structure of this thesis

This thesis is structured as follows: Chapter 2 provides a general introduction to maximum likelihood based inference of phylogenetic trees, and includes details on RAxML concepts that are a prerequisite for the rest of the text. In Chapter 3 we introduce methods to reduce the memory requirements of the PLF. The backbone algorithm for reducing the tree search space is described in Chapter 4. In Chapter 5 an introduction to GPU Programming is given, followed by Chapter 6, which describes our OpenCL-based GPU implementation of the PLF. In Chapter 7 we introduce `PUmPER`, our framework for perpetually updated phylogenies. Finally, we conclude and discuss future work in Chapter 8.

Chapter 2

Computational Molecular Phylogenetics

In this chapter we briefly introduce some basic concepts related to the field of Molecular Phylogenetics.

Phylogenetics is the study of evolutionary relationships among organisms that share common ancestors. The most common structure used to represent these relationships is a phylogenetic tree, which depicts ancestor-descendant relationships. From a biological perspective, it is known that evolution cannot be modelled as a strict branching process, because genetic information is not always transferred vertically. For instance, phenomena such as hybridization and lateral gene transfer (LGT) cannot be modeled by a phylogenetic tree. More complex structures, such as phylogenetic networks, are topics of active research [33]. In this thesis, however, we focus solely on phylogenetic trees.

The input data used to infer the trees can be morphological or molecular sequences. Morphological sequences are generated by encoding observed morphological traits. Molecular data (e.g., DNA, RNA, or AA) is obtained through sequencing technologies. Due to the continuously decreasing cost of Next-Generation Sequencing (NGS) technologies, molecular sequences are nowadays the most common source of data, especially for large-scale datasets. In this thesis, we will assume that molecular information, in particular DNA or protein data, is used in all the methods and applications discussed.

2.1 Statistical Models of Evolution

A molecular sequence (e.g., a DNA molecule), can be represented as a string of characters, where each character is an element of a finite alphabet. In the

case of DNA data the four *nucleotides* (also called *bases*) are `A, C, G, T`. For RNA data, these are `A, C, G, U`. For amino acid (AA) data, the alphabet comprises 20 characters that are also called *residues*. Each element of the alphabet represents a possible character state for a given sequence index (position or site).

A *statistical model of evolution* describes how likely it is for a state to mutate into a different state (transition) within a given evolutionary time t. We will now shortly describe nucleotide substitution models of evolution for DNA data. The distance between two DNA sequences is defined as the expected number of nucleotide substitutions per site. The number of substitutions can be estimated with a continuous-time Markov model. We assume that sites are evolving independently from each other. The Markov chain has four possible states (`A, C, G, T`) and is memory-less (the next state depends only on the current one). The *substitution rate matrix* $Q = \{q_{i,j}\}$ defines the probability that state i mutates into state j in an infinitely small amount of time dt. Each row of the Q matrix sums to zero. To compute the probability of a state mutating into another given time t, we need to compute the *transition-probability matrix* as follows:

$$P(t) = e^{Qt} \tag{2.1}$$

If Q is symmetrical, we call the model time-reversible. The Markov chain is reversible *if and only if* $\pi_i q_{i,j} = \pi_j q_{j,i}$ for any $i \neq j$, where π_i is the probability that the chain is in state i when time $t \to \infty$. The set $(\pi_A, \pi_C, \pi_G, \pi_T)$ is known as the *limiting distribution* or *stationary distribution* of the chain. If the states of the chain are in the stationary distribution, the chain will stay in that distribution. These state frequencies are also called *equilibrium base frequencies*. The set of equilibrium frequencies are determined by the Q matrix, but in practice is often estimated from the data or optimized numerically.

The numerical optimization of equilibrium base frequencies adds three free parameters to the model since π_A, π_C, π_G and π_T need to sum up to one. If equilibrium base frequencies are estimated from the data (by counting the number of occurrences of each state), they are called *empirical base frequencies*.

The standard approach to calculate the transition probability matrix $P(t)$ is to compute numerically the eigenvalues and eigenvectors of the rate matrix Q [112], as shown in Equation 2.2

$$Q = U \Lambda U^{-1} \tag{2.2}$$

where Λ is a diagonal matrix containing the eigenvalues of Q and U is

a square matrix whose ith column is the eigenvector of Q. Now we can rewrite Equation 2.1 as:

$$P(t) = e^{Qt} = Ue^{\Lambda t}U^{-1} \tag{2.3}$$

We can now describe the GTR (General Time Reversible) model [45], which is the most commonly used model for nucleotide substitution in large-scale phylogenetic inference. A detailed description of GTR and other nucleotide and amino acid evolution models can be found in Chapters 1 and 2 of [112].

$$Q = \{q_{i,j}\} = \begin{pmatrix} \cdot & q_{A,C} & q_{A,G} & q_{A,T} \\ q_{C,A} & \cdot & q_{C,G} & q_{C,T} \\ q_{G,A} & q_{G,C} & \cdot & q_{G,T} \\ q_{T,A} & q_{T,C} & q_{T,G} & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & a\pi_C & b\pi_G & c\pi_T \\ a\pi_A & \cdot & d\pi_G & e\pi_T \\ b\pi_A & d\pi_C & \cdot & f\pi_T \\ c\pi_A & e\pi_C & f\pi_G & \cdot \end{pmatrix} \tag{2.4}$$

Equation 2.4 can be decomposed into a product of a symmetric matrix of substitution rates and a diagonal matrix of equilibrium frequencies.

$$Q = \{q_{i,j}\} = \begin{pmatrix} \cdot & a & b & c \\ a & \cdot & d & e \\ b & d & \cdot & f \\ c & e & f & \cdot \end{pmatrix} \begin{pmatrix} \pi_A & 0 & 0 & 0 \\ 0 & \pi_C & 0 & 0 \\ 0 & 0 & \pi_G & 0 \\ 0 & 0 & 0 & \pi_T \end{pmatrix} \tag{2.5}$$

Substitution rates a, b, c, d, e, f are parameters that can be freely optimized (usually $f := 1$). The GTR has therefore a total of eight free parameters.

Other nucleotide substitution models can be derived from GTR by simply imposing restrictions on the base frequencies and/or the substitution rates. These derived models are more simple (nested within GTR) and have a lower number of free parameters. For instance, the Jukes-Cantor model [41] is given by $\pi_A = \pi_C = \pi_G = \pi_T = 0.25$ and $a = b = c = d = e = f = 1.0$.

2.2 Sequence Alignment

A *sequence alignment* can be seen as a matrix of n sequences of molecular data (DNA, RNA or protein), where each row corresponds to one sequence. In general, before sequences are aligned, each input sequence has a different length.

Any non-trivial alignment contains *gaps* (sites for which no state is assigned). Gaps may occur for biological reasons (character insertions or deletions). Data may also be undetermined due to technical reasons (data is

not read correctly during sequencing), or because information is missing (a specific gene for a specific taxon has never been sequenced). In the methods described in this thesis, missing and undetermined characters are treated as gaps.

Aligning sequences consists in identifying regions of common similarity, that is, gaps are inserted between characters so that the s columns in the matrix contain similar characters. Alignment algorithms strive to maximize this similarity according to a given optimization criterion. Gaps observed in an alignment column are interpreted as *indels* (insertions or deletions in the sequence). Columns including mismatches can be interpreted as sequence sites where one or several species have been subject of point mutations. In DNA data, point mutations are random SNPs (Single nucleotide Polymorphism), which usually take place during DNA replication.

If the codon resulting from the mutation change is translated into the same amino-acid, it is called a *silent* or *synonymous* substitution. Otherwise, if the mutation induces a change into a different amino acid, it is called *nonsynonymous substitution*, and depending on how different (in terms of biochemical properties) the new amino acid is, the protein function can be affected.

Sequence alignment is based on the notion that similar regions share a common evolutionary history. The interpretation is that regions with high similarity do not show changes because they correspond to an important, evolutionary conserved, structural property or function.

2.2.1 Pairwise Sequence Alignment

A pairwise sequence alignment is an alignment of two molecular sequences, that is, $n := 2$. In *global alignment*, every character of each sequence is aligned, that is, the alignment is at least as long as the longest sequence. In *local alignment*, a segment of one sequence, often called *query*, is aligned against a segment of a *reference* sequence.

The two main computational approaches are based on *dynamic programming* and *k-tuple* methods.

The Needleman–Wunsch algorithm [57], introduced in 1970, is based on dynamic programming and performs global alignment. In 1981, the Smith–Waterman algorithm [83], an adaptation of the Needleman-Wunsch algorithm, was introduced to solve the local pairwise alignment problem.

The importance of Needleman–Wunsch and Smith–Waterman is that, given a scoring function, they can find optimal solutions for the matching problem (by using dynamic programming) with time and space complexity given by $O(s_1 s_2)$, where s_i is the length of sequence i.

```
Sequence 1 for Gene arb AGCATCGATACCGATATCGCGAT
Sequence 2 for Gene arb AGAAGCGATCCCATACAGATGCGAT
Sequence 3 for Gene arb AGCATCGATACAGATGGCACGAT
Sequence 4 for Gene arb AGCAGCGATACATGGCACGAT

     <---- MSA for Gene arb ---->
Seq1 AGCATCGAT-----ACCGATATCGCGAT
Seq2 AGAAGCGATCCCATACAGATG---CGAT
Seq3 AGCATCGAT-----ACAGATGGCACGAT
Seq4 AGCAGCGAT-----AC--ATGGCACGAT
```

Figure 2.1: From a set of sequences to an alignment: Gaps are introduced in order to align the available sequences.

The Smith–Waterman algorithm is extensively used in applications such as, for instance, genome mapping and gene prediction, and has therefore been subject of intensive optimization and parallelization. Currently, there exist efficient implementations based on FPGAs (Field-programmable Gate Arrays) [2], GPUs [50], and vectorization on CPUs [71].

Word (k-tuple) methods are heuristic methods, and do not guarantee to find an optimal solution, but usually run faster than dynamic programming approaches. These methods are employed by tools that approximate matching queries against large sequence databases, such as BLAST [3] or FASTA [47]. Following a seed-and-extend strategy, initially exact matches of k-tuples (seeds) are identified. Then, the matching locations are extended into longer alignment candidates called HSPs (High Score Pairs). HSPs that are statistically significant may then be locally aligned with Smith–Waterman.

2.2.2 Multiple Sequence Alignment

A multiple sequence alignment (MSA) is an alignment of n molecular sequences, where $n > 2$. An alignment has s alignment columns, which are also often referred to as *sites*. The alignment length typically refers to the number of sites. The phylogenetic signal present in an alignment generally increases with alignment length [56].

For illustration, an MSA with length of $s := 27$ sites, corresponding to a fictional gene *arb*, and 4 sequences is shown in Figure 2.1

```
     <-------- Gene arb -------->
<----- Gene arb2 ------>
Seq1 AGCATCGAT-----ACCGATATCGCGATA-TAAGCTA-CGT-AGTTGAGGGT
Seq2 AGAAGCGATCCCATACAGATG---CGATG-TAAGCAA-CGT-AGTTGAGGGT
Seq3 AGCATCGAT-----ACAGATGGCACGATG-TAGCCTA-CCC-AGTTTCGCGG
Seq4 AGCAGCGAT-----AC--ATGGCACGATG-TGGCCTA-CCC-AGTTGCGCGT
```

Figure 2.2: Alignments for gene arb (red) and gene arb (blue) are concatenated to build a multiple-gene MSA.

It is common to concatenate alignments from different sources (e.g., different genes) for the same set of taxa. Such large concatenated datasets can then be organized into *partitions*. For instance, each gene may correspond to a separate partition. In a Maximum Likelihood partitioned analysis, the evolutionary model is evaluated under a specific optimized set of parameters for each partition.

Concatenated alignments, as shown in Figure 2.2, where two partitions have been concatenated, are also commonly referred to as supermatrices.

The sites that are put into one partition are often, but not necessarily, genes. The partition boundaries can also be based on other criteria such as codon positions. It is important to note that there is a distinction between *gene* and *species* phylogenies. Due to biological events such as gene loss, gene duplication and incomplete lineage sorting, a *gene* phylogeny may show a number of topological conflicts with the *species* phylogeny, which, therefore, is much more difficult to reconstruct. In this thesis, we always refer to a phylogeny as the tree we reconstruct given the alignment.

The process of building a multiple sequence alignment for phylogenetic inference is composed of several phases, that we summarize below:

1. Selection of the region of interest: The region may be a specific gene, a set of genes or even a genome (phylogenomics). This determines the length of the alignment.

2. Selection of a taxonomic group and sequence retrieval: The group of sequences (taxa) that will be included in the alignment. This step involves *homology* identification (two sequences are homologous if they share a lineage and have descended from a common ancestor). These sequences may be obtained through sequencing or querying databases such as GenBank [6].

3. Sequence alignment. Sequences are aligned and gaps are introduced so that a scoring criterion is maximized (matching sites are favoured, insertions of gaps are penalized). Sequence alignment of n sequences (or taxa) has been shown to be NP-hard [21], that is, the globally optimal solution is not computationally tractable. In practice, a heuristic search strategy called *progressive alignment* is often employed. Some popular multiple sequence aligners are *mafft* [42] and *muscle* [19].

The above procedure can be (partially) automated by computational pipelines such as `PHLAWD` [81]. The quality of the generated alignment, however, is difficult to assess, because the true alignment is unknown. The quality of the MSA has a substantial impact on the accuracy of the resulting phylogenetic reconstruction, and in general we assume that the given MSA is "correct". Nonetheless, it is important to be aware that, no matter how accurate the phylogenetic search method may be, a good phylogenetic tree can only be inferred if a good MSA is available. Unfortunately, there is no straight forward mechanism to assess the quality of a MSA. However, given a pairwise scoring function (see Subsection 2.2.1), it is possible to score an MSA computing the *sum-of-pairs* (SP) function. The SP is given by the sum of all column scores in the alignment, where the the score of each column is the sum of pair scores for all pairs that occur in the column. The accuracy of the phylogenetic method is hard to evaluate because the true evolutionary tree is unknown.

Furthermore, available high-quality database of curated sequence alignments, such as Pandit [107], can be used to benchmark MSA tools.

2.3 Phylogenetic Trees

In graph theory, a graph is a representation of a set of vertices where some pairs of vertices are connected by edges. In an undirected graph, edges have no orientation. A path is a sequence of edges that connect a sequence of vertices. A tree is an undirected graph in which any two vertices of the tree are connected by *exactly one* path. Therefore cycles, where two vertices are connected by more than one path, are not present in a tree.

In phylogenetics, such trees are called *phylogenetic trees*. The branching pattern defines the *topology*. Vertices are usually called nodes. Nodes are also called TUs (taxonomic units). Edges are referred to as *branches*. Terminal nodes of the tree are often called *tips*, *taxa*, or *OTUs* (Operational Taxonomic Unit). A single tip is called *taxon*, or OTU. The taxa represent living (extant) organisms for which molecular data *is* available and can be sequenced. The inner nodes represent hypothetical extinct common ancestors.

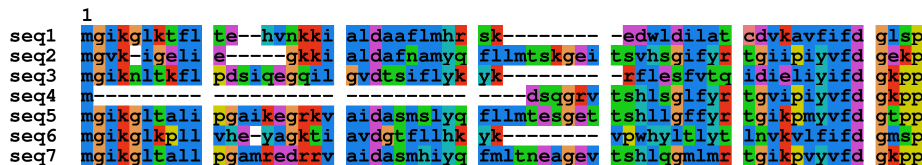

Figure 2.3: A protein alignment comprising seven sequences. Image generated with seaview version 4.3.1

In general, a phylogenetic tree represented as an *unrooted* binary tree, where all nodes have degree one (leaves) or three (inner nodes). Thus, given n taxa, there exist $n-2$ inner nodes (ancestral nodes) and $2n-3$ edges (branches). For example, Figure 2.4 shows an unrooted parsimony tree for the protein alignment shown in Figure 2.3.

The number of distinct possible rooted and unrooted topologies are given by Equation 2.6 [20].

$$N_{unrooted} = \frac{(2n-5)!}{2^{n-3}(n-3)!} \quad n \geq 3. \tag{2.6}$$

A *root* is a node from which a unique path leads to any other node of the tree. If a root is given in the tree, the direction of the evolutionary path is uniquely determined. Such topologies are called *rooted trees*.

The number of possible rooted topologies for n tips can be computed as the number of unrooted topologies for $n+1$ tips and is given by Equation 2.7.

$$N_{rooted} = \frac{(2n-3)!}{2^{n-2}(n-2)!} \quad n \geq 2. \tag{2.7}$$

In a rooted tree, a *clade* is a group of nodes that have evolved from a common ancestor.

The length of the branches represents the evolutionary distance between two nodes in the tree. Lengths of branches are measured in units of evolutionary changes. A tree without branch length values is called a *cladogram*. For example, Figure 2.5 shows an unrooted tree for the protein alignment shown in Figure 2.3.

2.4 Tree Reconstruction Methods

There exist different methods for reconstructing trees from molecular sequences. The common principle is the analysis of the differences among

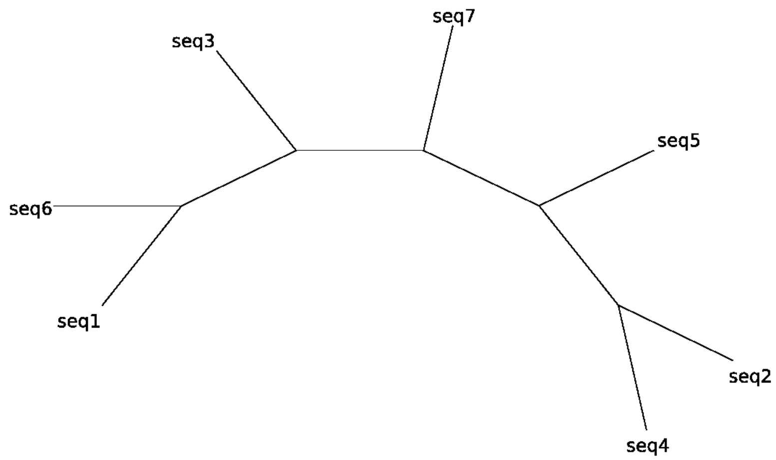

Figure 2.4: An unrooted phylogenetic tree based on the alignment from figure 2.3.

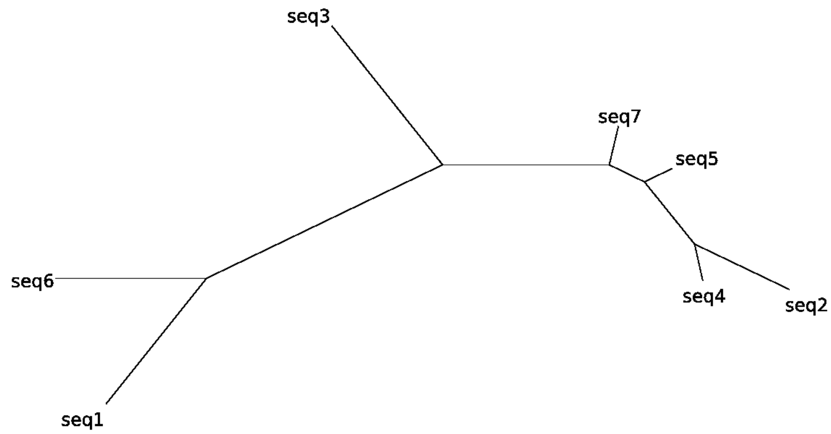

Figure 2.5: An unrooted tree with branch lengths based on the alignment from figure 2.3.

sequences and the use of that information to reconstruct an evolutionary history.

We can classify methods into *distance based* and *character based* methods.

Distance methods, where a *distance matrix* is computed from the pairwise distances among sequences, are based on the notion that common ancestry is reflected by sequence similarity. Once a distance matrix is available, a clustering algorithm is applied to deterministically generate a phylogenetic tree. A representative example of this group is Neighbour Joining(NJ). The canonical NJ problem is an $O(n^3)$ time algorithm and uses $O(n^2)$ space, where n is the number of species. This can lead to expensive memory requirements when the number of species is large, but efficient implementations, such as NINJA [106], have been designed to handle such datasets.

We now introduce character based methods, where different tree topologies are searched and evaluated under a given scoring criterion.

Parsimony methods compare trees based on the parsimony criterion. The *parsimony score* of a given tree is the minimum number of evolutionary state changes required to generate the MSA (Multiple Sequence Alignment) given the tree under evaluation.

Theoretically, in order to find the most parsimonious tree, all possible topologies should be evaluated. In practice, because the number of possible topologies grows super-exponentially with the number of taxa, the objective is to search the tree space for local optima by applying heuristics to evaluate a promising subset of topologies.

Maximum Likelihood (ML) and *Bayesian* methods use a Multiple Sequence Alignment and a statistical evolutionary model. Both methods are based on statistical frameworks that extensively rely on the computation of the Phylogenetic Likelihood Function (PLF). In Maximum Likelihood, the score to maximize is the likelihood, which is the probability of observing the MSA given the hypothetical tree and other parameters. Thus, we strive to explore and find the tree and set of parameters that maximize this score for our MSA. Likelihood scores cannot be compared across different MSAs.

Under Bayesian methods, the computed score is the (posterior) probability of the hypothetical tree, given the data (MSA and model). While this score has a clearer statistical interpretation (the score is the probability that the tree is the true tree), it also entails challenges such as, for instance, how to set the prior probability of a tree, which is unknown. Like Parsimony-based methods, Maximum Likelihood and Bayesian methods explore a subset of possible topologies using heuristics.

Distance Matrix methods are usually the fastest, since they do not require scoring alternative hypothesis during tree search. Parsimony trees are much faster to score than Likelihood/Bayesian trees, because they rely on a simpler

score calculation.

The accuracy of phylogenetic inference methods can be evaluated by conducting simulations, for instance, with tools like INDELible [26]. Such experiments allow to generate simulated true alignments (MSAs) and true trees. The simulated alignment can then be used to reconstruct trees by different methods. Then the accuracy of each method can be evaluated by computing the topological distance between the reconstructed tree and the simulated (true) tree. Trees on simulated data, however, tend to be easier to reconstruct than on real data. One possible explanation is that simulation programs generate the data using the same statistical models of evolution as the programs used to infer the tree, that is, the evolutionary model is given a priori [94]. Thus, accuracy results derived from such experiments should be interpreted with caution.

In general, distance methods are the least accurate, but also the fastest. In terms of accuracy, Likelihood and Bayesian methods clearly outperform other methods mainly due to the fact that they have an explicit evolutionary model [52]. In particular, Maximum Likelihood is consistent in the statical sense: the reconstructed tree converges to the true tree as the number of aligned sites goes to infinity [25]. In this thesis, we focused on Maximum Likelihood methods.

2.5 Computing the Likelihood of a Tree

As a consequence of the Maximum Likelihood search process, applying the PLF (Phylogenetic Likelihood Function) to evaluate alternative tree topologies dominates both running time and memory requirements of phylogenetic inference programs [39]. In Chapter 3 we describe the memory requirements in detail. We will now discuss how to compute the PLF.

The computation of the PLF relies on the following assumptions:

1. Sites in the MSA evolve independently from each other.

2. Evolution in different parts of the tree is independent.

3. A comprehensive unrooted phylogeny T, including a set of branch lengths b_{ij}, is given.

4. An evolutionary model is available, and defines the transition probabilities $P_{ij}(b)$, that is the probability that j will be the final state at the end of a branch of length b, given that the initial state is i.

5. The evolutionary model is time-reversible, that is, $\pi_j P_{ij}(b) = \pi_i P_{ji}(b)$

These assumptions allow us to compute the likelihood of the tree as the product of the site-likelihoods of each column i of the MSA with s columns, as shown in Equation 2.8.

Let T be an unrooted binary tree with n tips. Let θ be a set of (optimized or given) evolutionary model parameters (see Section 2.1). Let $\phi = \{b_{xy}\}$ be a set of (optimized or given) branch length values for tree T, where b_{xy} is the branch length value connecting nodes x and y in tree T ($b_{xy} = b_{yx}$, $x \neq y$ and $|\phi| = 2n - 3$).

$$L_T = \Pr(D \mid T, \theta, \phi) = \prod_{i=1}^{s} \Pr(D_i \mid T, \theta, \phi) \tag{2.8}$$

In order to prevent numerical underflow, it is common practice to compute and report log likelihood values:

$$log(L_T) = log(\Pr(D \mid T, \theta, \phi)) = \sum_{i=1}^{s} log(\Pr(D_i \mid T, \theta, \phi))$$

The root of the tree is, in general, unknown. In order to compute the likelihood on unrooted topologies, a virtual root is placed into an arbitrary branch. Because the substitution model is time-reversible, the likelihood of the tree is identical, independently of the branch chosen to place the virtual root. Figure 2.6 shows a virtual root placed on an arbitrary branch of the tree depicted in Figure 2.5.

For notation clarity, let us assume that we are working with a rooted tree and DNA data, where we have an alphabet of size four. Thus, only four states are possible and correspond to the nucleotides `A`, `C`, `G`, and `T`.

For each site `i`, four entries must be computed to store the *conditional likelihood* for nucleotide states `A`, `C`, `G`, and `T` at node p. The conditional likelihood entry $L_S^{(p)}(i)$ is the probability of everything that is observed from node p on the tree on up, at site i, conditional on node p having state S [25]. We can define the ancestral probability vector(APV) at node p and site `i` as:

$$\vec{L}^{(p)}(i) = (L_A^{(p)}(i), L_C^{(p)}(i), L_G^{(p)}(i), L_T^{(p)}(i)) \tag{2.9}$$

Let us consider Equation 2.10. Its derivation is explained in detail in Chapter 4 of [112].

$$L_A^{(p)}(i) = \Big(\sum_{S=A}^{T} P_{AS}(b_{qp}) L_S^{(q)}(i)\Big)\Big(\sum_{S=A}^{T} P_{AS}(b_{rp}) L_S^{(r)}(i)\Big) \tag{2.10}$$

This equation computes the ancestral probability vector (APV) entry $\vec{L}_A^{(p)}$ for observing the nucleotide `A` at site i of a parent node p, with two child nodes

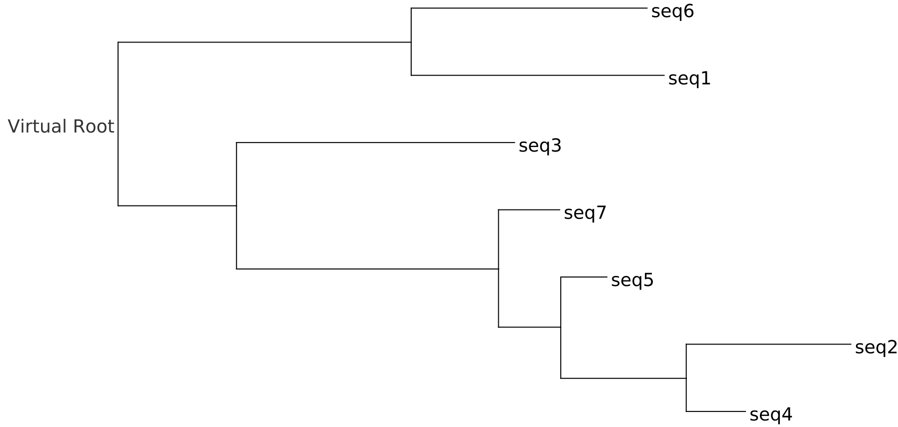

Figure 2.6: A rooted view of the unrooted ML phylogenetic tree from Figure 2.5. The virtual root is placed in an arbitrary branch.

q and r given the respective branch lengths b_{qp} and b_{rp}, the corresponding transition probability matrices $P(b_{qp})$, $P(b_{rp})$, and the probability vectors of the children $\vec{L}^{(q)}$, $\vec{L}^{(r)}$ for site i. The children/parent relationship is given by the position of the virtual root (see Figure 2.7).

The tips, which do not have any child nodes, must be initialized. Probability Vectors at the tips are also called *tip vectors*. In general, the sequence at the tips already have a known value for each site and therefore can be directly assigned a probability. For instance, if site `i` is an A, the tip vector can be directly initialized as: $(L_A^{(p)}(i), L_C^{(p)}(i), L_G^{(p)}(i), L_T^{(p)}(i)) := (1.0, 0.0, 0.0, 0.0)$.

To efficiently calculate the likelihood of a given, fixed tree, we execute a depth-first post-order tree traversal that starts at the virtual root. Via the post-order tree traversal, the probabilities at the inner nodes (ancestral probability vectors) are computed bottom-up from the tips toward the virtual root. This procedure to calculate the likelihood is called Felsenstein pruning algorithm [23].

At the virtual root node, the site likelihood of the tree can be computed using equation 2.11

$$L_T(i) = \Pr(D_i \mid T, \theta, \phi) = \sum_{x=A}^{T} \pi_x L_x^{(root)}(i) \tag{2.11}$$

In Equation 2.11, π_x is the equilibrium frequency for state x. It represents the probability that the nucleotide at the root is x.

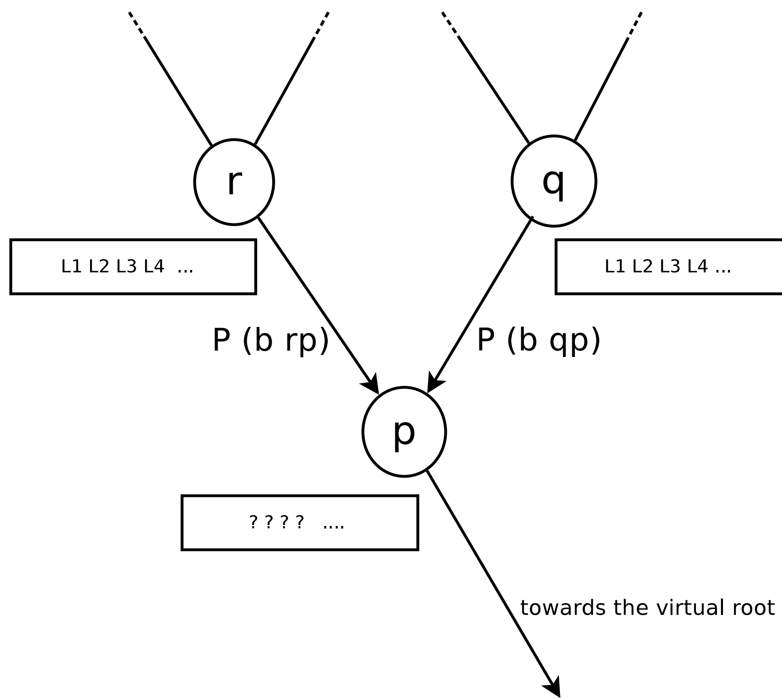

Figure 2.7: Ancestral probability node entries are computed based on the APVs of the children nodes, the model and the transition probability matrices $P(b_{qp})$ and $P(b_{rp})$.

A more detailed description of PLF computations and efficient PLF implementations can be found in [88].

2.5.1 Accounting for rate heterogeneity

An important extension in the computation of the PLF is that not all alignment sites evolve at the same rate [48]. Ignoring rate variation may result into incorrect reconstruction of phylogenies [111]. Thus, models of *rate heterogeneity* have been introduced in order to accommodate the biological phenomenon of rate variation .

The most common model is the Γ [110] model of rate heterogeneity, where for each site the likelihood is integrated over a continuous Γ distribution of rates. The gamma density is given by Equation 2.12.

$$g(r;\ \alpha, \beta) = \frac{\beta^{\alpha} r^{\alpha-1} e^{-\beta r}}{\Gamma(\alpha)} \tag{2.12}$$

The mean is given by α/β. The Γ model sets $\alpha := \beta$, so that the mean is 1. Thus, the distribution has only one parameter α, which is usually adjusted by numerical optimization. Figure 2.8 shows the effect of α on the distribution of rates. If $\alpha < 1$, the distribution implies that there is a large amount of rate variation, that is, many sites evolve slowly but some sites may have high rates. Large values of α tend to minimize the effect of rate variation, since most rates fall close to the mean.

We can adapt Equation 2.11 to compute the likelihood of the tree integrating over the rate distribution.

$$L_T(i) = \Pr(D_i \mid T, \theta, \phi, \alpha) = \int_0^{\infty} g(r;\ \alpha) \sum_{x=A}^{T} \pi_x L_x^{(root)}(i, r)\, dr \tag{2.13}$$

In phylogenetic software tools, a more computationally tractable *discrete gamma model* is implemented. This model approximates the continuous distribution using K rate categories with equal probability. Accounting for rate heterogeneity increases accuracy but has an important impact on performance. For every node, site and state K entries must be computed in the APVs. Therefore, likelihood computations and memory requirements increase by a factor of K. The mean rate in each category is used to represent all the rates in that category. The most common choice is to use four rates with $K := 4$, which is sometimes called $\Gamma 4$.

Individual per-site evolutionary rates are not used because this might lead to over-fitting and over-parametrization of the data. The *PSR* model

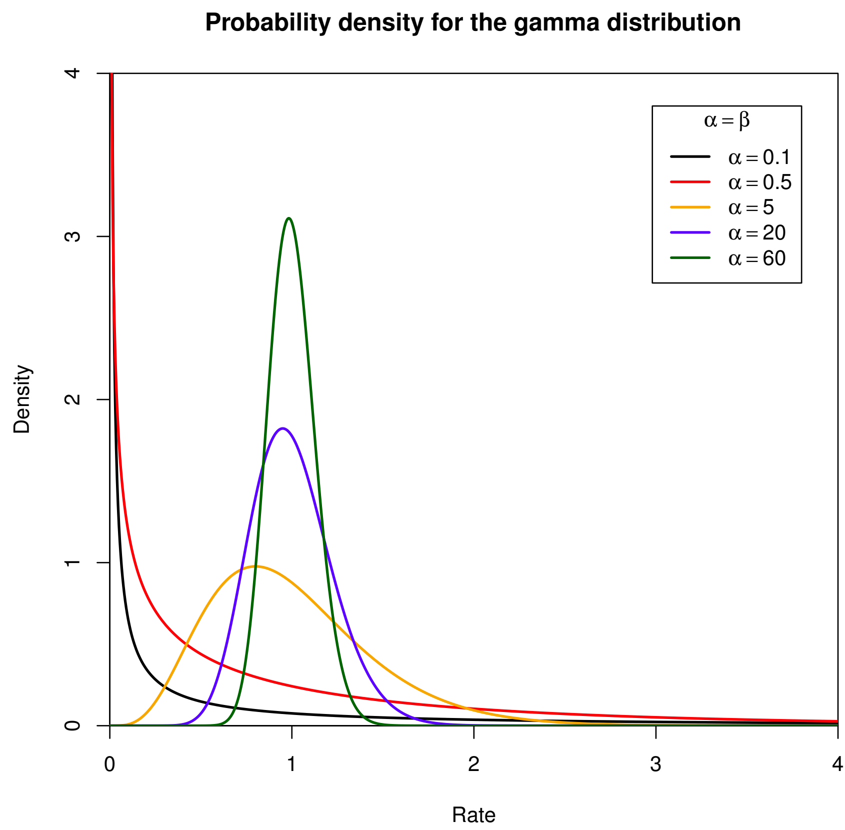

Figure 2.8: The shape parameter α is inversely related to the rate variation among sites: Larger values of α reduce the variation.

(previously known as CAT [86]) attempts to alleviate this issue by using a fixed number $c << s$, where s is the number of sites and c is the number of rate categories. Each individually optimized evolutionary rate r_i, with $1 < i <= s$, is mapped to a rate category ρ_j, with $0 < j < c - 1$. The relevance of the PSR model is that, while memory usage and number of computations are four times lower than with the $\Gamma 4$ model, reconstructed trees show comparable Γ-likelihood values [86].

2.6 Maximum Likelihood Tree Search

The objective of *ML (Maximum Likelihood) phylogenetic inference* is to find the topology that maximizes the (log) likelihood score (computed by the PLF), given the data (MSA) and a statistical model of evolution.

A naïve strategy for tree space exploration is an exhaustive search, that is, all possible unrooted topologies are enumerated. Then, for each topology, the branch lengths and model parameters are optimized to maximize the likelihood. The topology with the highest likelihood is the optimal ML topology. In practice, this is only feasible for very small trees. In general, finding the optimal ML topology is NP-hard [70].

Tree space is explored making use of heuristics, with the goal of scoring only a small subset of good topologies. One idea for exploring tree space is to use a greedy algorithm. First, a comprehensive tree is generated randomly or with a faster method such as Neighbour Joining or Parsimony. Then, small rearrangements are applied to the topology of this initial tree, generating new trees that can be scored with the PLF. Whenever a new tree is better (higher likelihood score), it is kept and new rearrangements are applied to this tree.

Typical *topological moves* for finding/reconstructing a tree topology with an improved likelihood score are SPR (Subtree Pruning and Re-grafting), NNI (Nearest Neighbor Interchange) or TBR (Tree Bisection and Reconnection) moves.

Most of the time, as the SPR example in Figure 2.9 shows, these moves only induce local changes to the tree topology. In other words, the majority of the ancestral probability vectors are not affected by the topological change and do therefore not need to be recomputed/updated with respect to the locally altered tree topology via a full post-order tree traversal (see Section 2.5). In these cases, we can compute the likelihood of the tree after a topological move by *only* updating the (mostly small number of) ancestral probability vectors included a local post-order tree traversal.

All standard ML-based programs deploy search strategies that typically

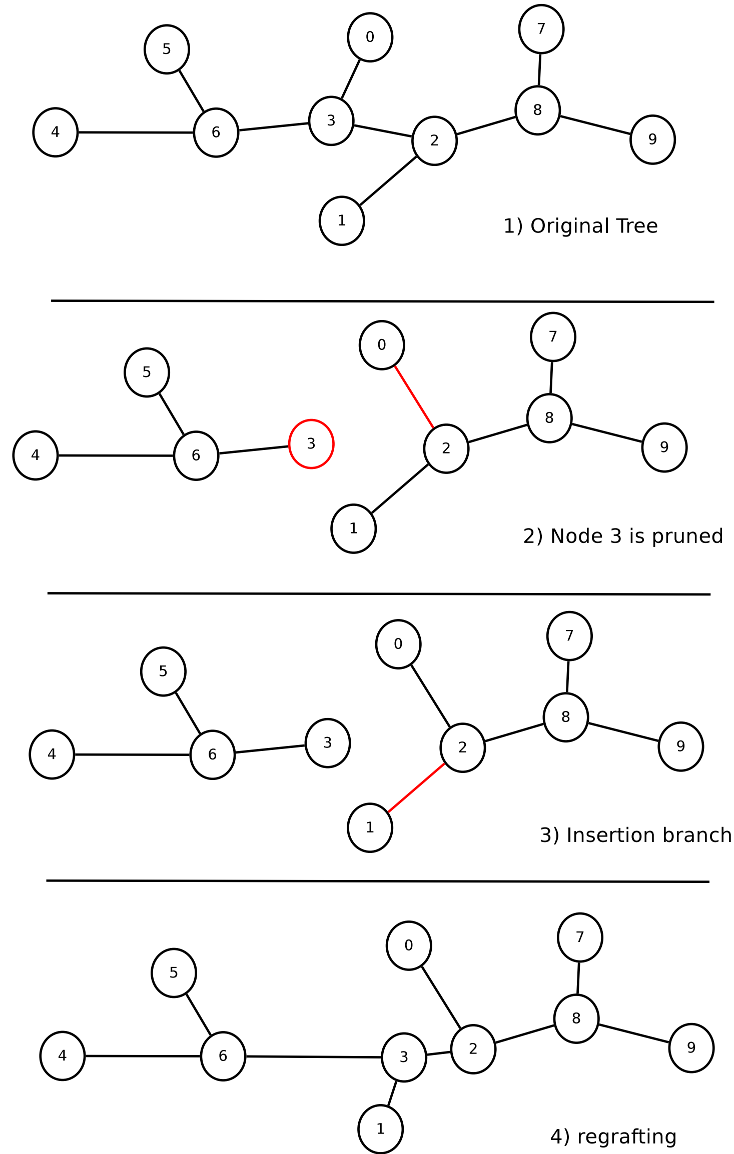

Figure 2.9: SPR (Subtree pruning and regrafting) moves can be divided in stages. In (1) a comprehensive tree is available. (2) a subtree defined by node 3 is selected and pruned. A new branch length needs to be computed to reconnect nodes 0 and 2. (3) The algorithm selects a insertion branch, usually close to the point of pruning. (4) The pruned subtree is reinserted and new branch lengths are recomputed. In *lazy* SPRs like the one shown here, only surrounding branch lengths are re-estimated.

require updating only a small fraction of probability vectors in the vicinity of the topological change.

The phylogenetic search is finished when no better topology can be found according to a *convergence criterion*. There are several possible convergence criteria, which vary among different phylogenetic software implementations. For instance, in PhyML, search convergence is determined by NNIs [29]. RAxML [87] bases its criterion on SPRs.

The best tree after convergence is mostly a local optimum. There is no guarantee, however, that the global optimum can be found. Therefore the usual practice is to run several searches with different starting trees to explore the landscape of local optima.

Assessment of Phylogenetic Trees

In general, the true evolutionary history is unknown. Therefore, an important questions is how to assess the significance of a tree, and how to quantify how different a set of trees is among itself.

A *bipartition*, also referred to as split, represents a branch of the tree, and defines two disjoint sets of taxa. An unrooted topology is uniquely defined by its set of bipartitions. A non-trivial bipartition is a bipartition where both sets contain more than one element. A trivial bipartition is a bipartition where one set contains one element.

For example, given the tree `((A,B),((C,D),E))`, there are two non-trivial bipartitions defined by `{A,B},{C,D,E}` and `{A,B,E},{D,C}`.

One common approach to measure how well a tree represents the underlying data is to repeat the phylogenetic search on resampled datasets and quantify how often a bipartition is recovered.

Given a Maximum Likelihood tree, bootstrap analysis [24] can be conducted to resample the data. First, bootstrap replicates are generated by resampling columns of the original MSA with replacement. Thereafter, a phylogenetic is tree inferred on each bootstrap replicate MSA, using the same inference method as for the maximum likelihood tree. Finally, for each bipartition of the maximum likelihood tree, a *bootstrap support value* is computed as the percentage of bootstrap trees that contain that bipartition. Bootstrap support values reflect how well the maximum likelihood tree represents the underlying data. Bootstrap replicate trees can also be summarized into a single *consensus tree*. For instance, the majority rule (MR) criterion yields a consensus tree which only includes bipartitions that exist in half or more of the replicate trees.

Topological distances between unrooted trees are often measured with the *Robinson-Foulds distance* [69]. The Robinson-Foulds distance (RF dis-

tance) between two trees, also termed *symmetric difference* and *partition metric*, is defined as the number of bipartitions that are not shared between two trees. This value is usually normalized by the total possible number of non-trivial bipartitions. Thus, identical trees have an RF distance 0. Trees that only share trivial bipartitions have an RF distance 1.

Phylogenetic trees can be considered as competing evolutionary hypothesis. Multiple plausible hypothesis (trees) can be compared with site-likelihood based statistical tests such as the Kishino-Hasegawa (KH) [43], the Shimodaira-Hasegawa (SH) [77] and the Approximately Unbiased (AU) tests [76]. Given the site-likelihood values of each tree, the software package CONSEL [78] can compute p-values for each of these tests. A confidence set of trees (set of trees that are significantly better than the others) can be obtained by collecting the trees with p-values that are larger than a chosen significance level, for instance, trees with $p > 0.05$.

2.7 Phylogenetic Likelihood Library

The *Phylogenetic Likelihood Library* (PLL) [27] is a parallelized and highly optimized software library for phylogenetic calculations. It has been derived from the highly tuned PLF implementation of RAxML [87].

The PLL library comprises implementations of state-of-the-art algorithms and data structures, including those described in Section 2.8, along with low-level technical and hardware-dependent optimizations. The library can calculate (and optimize) the likelihood on a phylogenetic tree for different statistical models and data types. It also implements branch length optimization, and various tree rearrangement techniques, such as Subtree Pruning and Regrafting (SPR) and Nearest Neighbor Interchanges (NNI). Moreover, the PLL can use multiprocessor architectures via a fine-grain parallelization of the PLF that relies on PThreads or MPI. In the parallel version, the alignment sites (or alignment partitions) are distributed across processors. Single x86 cores use SSE3 or AVX intrinsics to accelerate computations.

While the PLL is currently work in progress, it has already been successfully deployed to substantially accelerate DPPDIV [31], a Bayesian program for divergence time(s) estimates. It has also been used to implement the proof-of-concept implementation on GPUs described in Chapter 6.

2.8 RAxML-family implementation concepts

In this Section, we describe some implementation details regarding data structures and algorithms that are required for phylogenetic computations. The data structures and terminology we present here resemble the ones used in the RAxML-family, which includes the original `standard-RAxML` [87], as well as more recent phylogenetic codes such as `RAxML-light` [91], `ExaML` [90], and the `PLL` library [27].

Thus, we introduce some generic terminology, including pseudo-code for some functions and data structures whose naming and implementation may slightly differ but serve the same functionality. Most of these concepts are also present in other state-of-the-art phylogenetic codes.

These implementation details are required to outline the techniques described on Chapter 3, Chapter 4 and Chapter 6 .

2.8.1 Node records

The *node record* is the fundamental data structure to compute the likelihood of a tree. Each node record can be seen as a subtree root. The data structure in Pseudo-code 1 represents a node record.

Pseudo-code 1 The node record data structure

```
typedef struct noderec
{
  double          z;        /* branch length value */
  struct noderec  *next;
  struct noderec  *back;
  int             number;   /* tree node identifier */
  char            x;        /* true if towards the root */
} node;
```

2.8.2 Internal tree nodes

The internal nodes of a binary tree are represented by circular lists of three node records. Each internal node of the tree corresponds to one ancestral probability vector (APV). However, the entries of the APV differ depending on the orientation of the node. The internal nodes of the tree are circular lists composed of three node records, where, by definition, the condition `x == TRUE` holds only for one node record. In Figure 2.10, tree node records

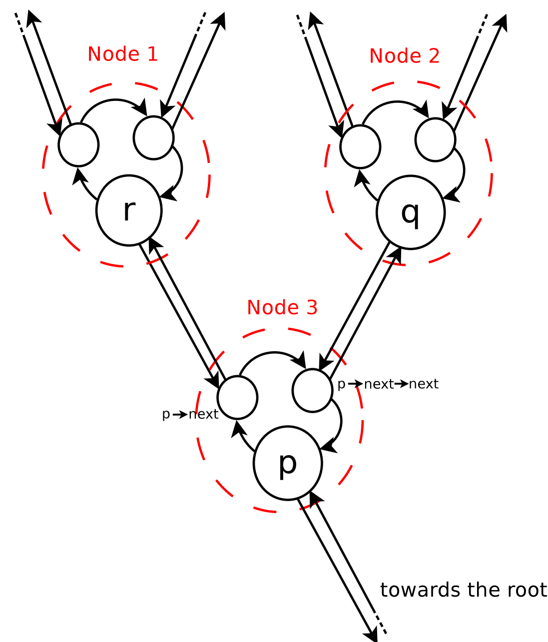

Figure 2.10: Circular double linked lists of node records (black) represent tree nodes (red). Back links are not shown. Each node record represents a possible subtree with different ancestral probabilities. For each tree node, the APV entries are only valid for the node record oriented towards the virtual root (p, q and r).

are labeled `p`, `q` and `r`. The ancestral probability vector for the node corresponds to the subtree root defined by the node record oriented towards the virtual root.

Thus, given a node record `p`, all node records share the same number, which identifies uniquely the internal node (`p->number == p->next->number`).

2.8.3 Data structure for Trees

The partition data (see Pseudocode 3) can be accessed through `tipSequences`, which point to the raw alignment data in memory. The array `tipVector` contains the product of the left eigenvectors for all possible states. It can be used as a look-up table to compute the likelihood entries at the tips.

The traversal descriptor (see Pseudocode 3) is used as a list of update operations for ancestral probability vectors in inner nodes. Each update operation involves two input child nodes (by convention `q` and `r`), and one parent output node (`p`). The `traversalInfo` data structure represents an update operation. The numbers `pNumber`, `qNumber`, and `rNumber` uniquely

Pseudo-code 2 The partition data structure

```
typedef struct
{
  double        **apvEntries;
  unsigned char **tipSequences;
  double        *tipVector;
} pInfo;
```

identify nodes `p`, `q`, and `r`.

The parent node is always an inner node of the tree. The children nodes can either be tips or inner nodes. Thus, there are three possible cases. The `tipCase` field stores the type of configuration (inner/inner, inner/tip or tip/tip).

Pseudo-code 3 The traversal-descriptor data structure

```
typedef struct{
  int tipCase;
  int pNumber;
  int qNumber;
  int rNumber;
  double qz;
  double rz;
} traversalInfo;

typedef struct
{
  traversalInfo *ti;
  int count;
} traversalDescriptor;
```

The tree data structure (see Pseudocode 4) includes the previously presented data structures.

2.8.4 Computing the likelihood on a Tree

For computing the likelihood on a tree, we need the following two core functions:

Pseudo-code 4 A simplified tree data structure

```
typedef struct
{
  node                 **nodep;
  node                 *start;
  pInfo                *partitionData;
  traversalDescriptor  *td;
} tree;
```

Pseudo-code 5 Building the traversal descriptor with a post-order traversal

```
void computeTraversalDescriptor(node *p, traversalDescriptor *td)
{
  if isTip(p->number)
    return;

  node *q = p->next->back;
  node *r = p->next->next->back;

  if(isTip(q->number) && isTip(r->number))
  {
    addEntry(td, p, q, r, TIP_TIP);
  }
  else if(isTip(q->number) || isTip(r->number))
  {
     swap(q,r) if isTip(r->Number);
     computeTraversalDescriptor(q, td);
     addEntry(td, p, q, r, TIP_INNER);
  }
  else
  {
     computeTraversalDescriptor(q, td);
     computeTraversalDescriptor(r, td);
     addEntry(td, p, q, r, INNER_INNER);
  }
}
```

The `newview(tree *tr, node *p)` function updates an ancestral probability vector (`apvEntries[p->number]`) given two child nodes and given two transition probability matrices P for the respective branch lengths leading to these child nodes (see Figure 2.7).

The `evaluate(tree *tr)` function is called at the virtual root that has been placed into the unrooted tree for scoring it. Given the two ancestral probability vectors at either end of the rooted branch and the branch length, `evaluate()` computes the overall log likelihood of the tree.

As described in Section 2.5, in order to compute the log likelihood of a tree, we need to conduct a depth-first post-order traversal of the tree topology (starting at the virtual root). Pseudocode 5 shows how the traversal descriptor is filled. Thereafter, we invoke `newview()` for each likelihood operation included in the descriptor. If we conduct a full traversal, the length of the descriptor will be equal to the number of inner nodes (all internal nodes are visited).

In some cases, for instance, while executing SPR moves (see Figure 2.9), only a partial traversal is required, because the local perturbation only requires a subset of the APVs to be updated. Once the traversal descriptor has been executed, all APVs are oriented towards the virtual root. Thus, we invoke `evaluate()` to calculate the overall log likelihood score of the tree.

2.8.5 Optimizing branch lengths

The functions `evaluate()` and `newview()` are sufficient to implement a Bayesian inference program, since the MCMC procedure, unlike the maximum likelihood method, does not require dedicated parameter optimization routines.

In the Maximum Likelihood (ML) framework however, we do need such explicit parameter optimization routines. Typically, branch length optimization is implemented via the Newton-Raphson procedure.

Branch length optimization accounts for approximately 20% to 30% of total execution time in state-of-the-art ML tree inference algorithms [10, 93]. To optimize a specific branch, we need to invoke `newview()` first on the nodes at either end of the branch to ensure that the ancestral probability vectors are oriented towards the branch that is being optimized. In fact, this corresponds to placing the virtual root of the tree into the branch that shall be optimized. Furthermore, we also need to invoke `newview()` when a branch has been changed to ensure that the ancestral probability vectors in the tree are in a consistent state and reflect the altered branch.

The Newton-Raphson branch length optimization procedure involves the following two routines:

- `coreDerivative()` computes the first and second derivative of the likelihood function at a given branch.

- `sumGAMMA()` pre-computes the element-wise product of the ancestral probability vectors to the left and the right of the branch under optimization. This product is then re-used repeatedly by iterations of `coreDerivative()` and allows to save time by avoiding redundant computations.

The RAxML-family codes provide a function for direct branch length optimization (called `makenewz()`) using the Newton-Raphson procedure that uses the two functions described above.

Chapter 3

Memory-Saving Techniques

The computation of the phylogenetic likelihood function (PLF) for reconstructing evolutionary trees from molecular sequence data is both memory- and compute-intensive.

Based on our interactions with the RAxML [87] user community, we find that, memory shortages are increasingly becoming a problem and represent *the* main limiting factor for large-scale phylogenetic analyses, especially at the genome level. At the same time, the amount of available genomic data is growing at a faster pace than RAM sizes. For instance, to compute the likelihood on a simulated DNA alignment with 1,481 species and 20,000,000 sites (corresponding roughly to the 20,000 genes in the human genome) on a single tree topology under a simple statistical model of nucleotide substitution within 48 hours, 1TB of memory and a total of 672 cores are required.

Some solutions to this problem have been previously presented. The use of single precision arithmetics [7] can reduce memory requirements by 50%, but it can also potentially introduce numerical instability. Novel algorithmic solutions [92] have been introduced to exploit large sections of missing data. Those approaches, however, remain dataset-specific, that is, their efficiency/applicability depends on the specific properties of the Multiple Sequence Alignment (MSA) that is used as input.

The concepts presented in this chapter can be used to exactly compute the phylogenetic likelihood function while significantly reducing memory requirements. These techniques can be applied to all programs that rely on the phylogenetic likelihood function and can contribute significantly to enabling the computation of whole-genome phylogenies. In Section 3.1, we discuss the memory requirements associated to the computation of the PLF, which dominate the memory requirements of a typical phylogenetics application. In Section 3.2, we present an out-of-core implementation of the PLF. Section 3.3 describes a technique to compute the PLF by trading memory for

Node v			
Site 1		Site 2	
Rate 0	Rate 1	Rate 0	Rate 1
$L_A L_C L_G L_T$	$L_A L_C L_G L_T$	$L_A L_C L_G L_T$	$L_A L_C L_G L_T$

Figure 3.1: Memory layout of an ancestral probability vector with a two-rate Γ model of rate heterogeneity.

recomputations of ancestral probability vectors. In Subsection 3.4.1, we discuss the reduction of memory and computations by means of subtree equality vectors when alignments with large gappy sections are present.

3.1 Memory requirements for the PLF

The PLF is defined on unrooted binary trees. The n extant species/organisms of the MSA under study are located at the tips of the tree, whereas the $n-2$ inner nodes represent extinct common ancestors. The molecular sequence data in the MSA that has a length of s sites (alignment columns) is located at the tips of the tree. The memory requirements for storing those n tip vectors of length s are not problematic, because one 32-bit integer is sufficient to store, for instance, 8 nucleotides when ambiguous DNA character encoding is used. The memory requirements are dominated by the ancestral probability vectors that are located at the ancestral nodes of the tree. Depending on the PLF implementation, at least one such vector (a total of $n-2$) will need to be stored per ancestral node. For each alignment site $i, i = 1...s$, an ancestral probability vector needs to hold the data for the probability of observing an `A`,`C`,`G` or `T`. Thus, under double precision arithmetics and for DNA data, a total of $(n-2) \cdot 8 \cdot 4 \cdot s$ bytes is required for the most simple evolutionary models. If the standard (and biologically meaningful) Γ model of rate heterogeneity [110] with 4 discrete rates is deployed, this number increases to $(n-2) \cdot 8 \cdot 16 \cdot s$, since we need to store 16 probabilities for each alignment site. Further, if protein data is used that has 20 instead of 4 states, under a Γ model the memory requirements of ancestral probability vectors increase to $(n-2) \cdot 8 \cdot 80 \cdot s$ bytes.

Figure 3.1 depicts a standard memory layout. For each alignment site, the ancestral probability vector contains 2 rate blocks, which correspond to a two rate discretization. In real-world applications four rates are typically used (see Section 2.5). Each rate block contains 4 entries (1 per state, denoted by L_A, L_C, L_G, and L_T).

Thus, total memory requirements can be easily estimated in advance,

since they are strongly dominated by the size of the ancestral probability vectors (APVs). The APVs requirements are directly proportional to alignment size.

3.2 Out-of-Core

The content of this Section has been derived from the following peer-reviewed publication:
F. Izquierdo-Carrasco and A. Stamatakis. Computing the phylogenetic likelihood function out-of-core. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 444–451, 2011

In this Section, we study the performance of an out-of-core execution of the phylogenetic likelihood function by means of a proof-of-concept implementation in standard RAxML.

In cases where the data structures for computing a function do not fit into the available Random Access Memory (RAM), out-of-core execution may be significantly more efficient than relying on paging by the Operating System (OS). This is usually the case, because application-specific knowledge *and* 'page' granularity can be deployed to more efficiently exchange data between RAM and disk. Since the PLF is characterized by predictable linear data accesses to vectors, as we show, PLF-based programs are well-suited to the out-of-core paradigm.

We find that, in our proof-of-concept implementation, RAM miss rates are below 10%, even if only 5% of the required data structures are held in RAM. Moreover, we show that our proof-of-concept implementation runs more than 5 times faster than the respective standard implementation when paging is used. The method is generally applicable. The source code and results are available at `http://www.exelixis-lab.org/web/personal_page/izquierdo/ooc.tar.gz`

The remainder of this Section is organized as follows: In Subsection 3.2.1 we briefly discuss related work in the general area of out-of-core computing and some applications to phylogenetic reconstruction using Neighbor Joining which exhibits substantially different data access patterns. In Subsection 3.2.2 we initially outline the necessary underlying principles of the PLF that allow for out-of-core execution and describe the optimization of the proof-of-concept implementation in RAxML. In the subsequent Subsection 3.2.3 we describe the experimental setup and provide respective performance results.

3.2.1 Related Work

The I/O bandwidth and communication between internal memory (RAM) and slower external devices (disks) can represent a bottleneck in large-scale applications. Methods that are specifically designed to minimize the I/O overhead via explicit, application-specific, data placement control and movement (e.g., between disk and RAM) are termed out-of-core algorithms (frequently also called: External-Memory (EM) algorithms; we will henceforth use the terms as synonyms).

EM data structures and algorithms have already been deployed for a wide range of problems in scientific computing including sorting, matrix multiplication, FFT computation, computational geometry, text processing, etc. Vitter provides a detailed review of work on EM algorithms in [105].

With respect to applications in phylogenetics, EM algorithms have so far only been applied to Neighbor-Joining (NJ) algorithms [79, 106]. NJ is fundamentally different from PLF-based analysis (see Chapter 2). NJ is a clustering technique that relies on updating an $O(n^2)$ distance matrix that comprises the pairwise distances of the n organisms for which an evolutionary tree is reconstructed. The size of this matrix becomes prohibitive for datasets with several thousand organisms. The data access pattern is dominated by searching for the minimum in the $O(n^2)$ distance matrix at each step of the tree building process.

3.2.2 Computing the PLF Out-of-Core

Data Access Patterns

The reason why the PLF is particularly well-suited for out-of-core execution is the regularity and predictability of data access patterns. The likelihood on the tree is computed according to the Felsenstein pruning algorithm [23], which we described in Section 2.5. Given an arbitrary rooting of the tree, one conducts a post-order tree traversal to compute the likelihood. The s values in an ancestral probability vector are computed recursively by combining the values in the respective left and right child vectors. Thus, such a tree traversal to compute the likelihood proceeds from the tips towards the virtual root in the tree. The ancestral probability vectors are accessed linearly and the ancestral probability vector access pattern is given by the tree topology.

In general terms, good I/O performance in EM algorithms is achieved by modifying an application such as to achieve a high degree of data locality. For the PLF, the straight-forward candidate data structure (the 'page') for transfers between disk and RAM are the ancestral probability vectors. In

RAxML they are stored linearly in memory. The replacement strategy simply needs to exploit the access pattern induced by the tree. In current ML search algorithms, the tree is not entirely re-traversed for every candidate tree that is analyzed. A large number of topological changes that are evaluated are local changes. Thus, only a small fraction of ancestral probability vectors needs to be accessed and updated for each tree that is analyzed.

The typical minimum HW block size is 512 bytes, although some operating systems use a larger block size of 8KB [105]. For the PLF this granularity is not an issue, since a representative ancestral probability vector is significantly larger than the block size. For instance, consider a typical, but still comparatively small, MSA of DNA data with length $s = 10,000$ and $n = 10,000$ species. To compute the PLF, $9,998$ ancestral probability vectors need to be stored. Each of these vectors is stored contiguously in memory and has a size of $10,000 \cdot 8 \cdot 4 \cdot 4 = 1,280,000$ bytes (1.28MB) under double precision arithmetics for a Γ model of rate heterogeneity with 4 discrete rates.

Thus, we can simply set the logical block size b (i.e., the 'page' size) to the size of an individual ancestral probability vector. Therefore, I/O operations can be amortized, that is, each read or write to disk will access a contiguous number of bytes on disk that is significantly larger than the minimum block size.

Basic Implementation

In our basic implementation, we store all ancestral probability vectors that do not fit into RAM contiguously in a single binary file (see figure 3.2). We deploy an appropriate data structure to keep track of which vectors are currently available in RAM and which vectors are stored on disk.

Let n be the number of ancestral probability vectors and m the number of vectors in memory, where $m < n$ (i.e., $n - m$ vectors will be stored on disk). Due to the way the likelihood is computed by combining the values of two child vectors for obtaining an ancestral probability vector (see Section Section 2.5 and Figure 2.7), we must ensure that $m \geq 3$. In other words, the space allocated in RAM must be large enough to hold at least three ancestral probability vectors. To allow for easy assessment of various values of m with respect to n, we use a parameter f that determines which fraction of required RAM will be made available, that is, $m := f \cdot n$. Now, let w be the number of bytes required for storing an ancestral probability vector, that is, our proof-of-concept implementation will only allocate $m \cdot w$ bytes. We henceforth use the term *slot* (one may think of this as a page), to refer to a segment of available memory (an ancestral probability vector) with a size of

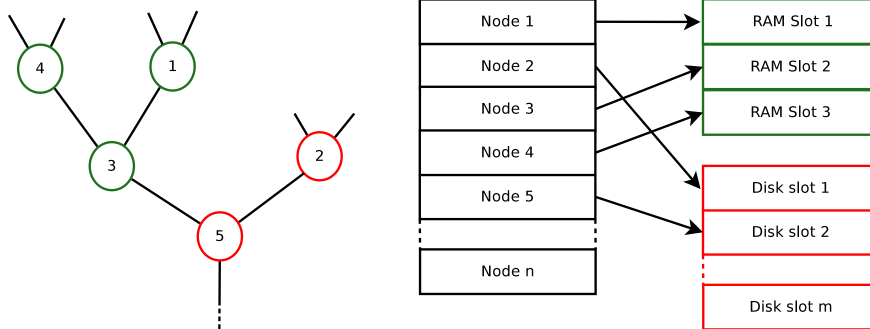

Figure 3.2: Each numbered ancestral node in the tree (left part of figure) needs to hold a vector of doubles. These vectors are either stored on disk (red) or in memory (green). A data structure (middle part of figure) is used to look up the memory location (right part of figure) of each ancestral vector. Each slot has the size of an ancestral probability vector.

w bytes.

To orchestrate data transfers and to control the location of vectors, we use the following C data structure (unnecessary details omitted).

```
typedef struct
{
  FILE                 **fs;
  unsigned int         num_files;
  size_t               slot_width;
  unsigned int         num_slots;
  unsigned int         *item_in_mem;
  unsigned int         num_items;
  unsigned int         disk_items;
  nodemap              *itemVector;
  double               *tempslot;
  boolean              skipReads;
  replacement_strategy strategy;
}map;
```

The array `itemVector` is a list of n pointers (see Figure 3.2) that is indexed using the (unique) ancestral node numbers. Each entry of type `nodemap` keeps track of whether the respective ancestral vector is stored on disk or in memory. More specifically, if the ancestral vector resides in RAM,

we maintain its starting address in memory. If the vector resides on disk, we maintain an offset value for its starting position in the binary file.

We also maintain an array of n integers (`item_in_mem`) that keeps track of which vector is currently stored in which memory slot.

Replacement Strategies

In the standard implementation of RAxML, where $n = m$, all vectors are stored in RAM. Whenever an ancestral probability vector is required, we simply start reading data from the respective starting addresses for node `i` that is stored in the address vector `xVector[i]`. In the out-of-core version, we use a function `getxVector(i)`, which returns the memory address of the requested ancestral vector for node `i`. The entire out-of-core functionality is transparently encapsulated in function `getxVector(i)`. The function will initially check, whether the requested vector is already mapped to RAM. If not, it will determine an adequate memory slot (according to some replacement strategy, see below), and swap the currently stored vector in that slot with the requested vector in the binary file.

A constraint for the replacement strategy is that, we must ensure that the 3 vectors required to compute the values at the current ancestral node `i` (vector `i` and the two child nodes `j` and `k`) reside in memory. Using the example in Figure 3.2, let us assume that we are traversing the tree and that the virtual root is located in the direction of the dotted line. When we need to compute the values for vector 3 (`i`), vectors 1 (`j`) and 2 (`k`) need to reside in memory. Calling `getxVector(1)` will return the address of memory slot#1 (where the vector for node 1 is already available). However, vector 2 may be located on disk. A call to `getxVector(2)` will thus require a swap of vectors, but slots #1 and #3 must be excluded (pinned to memory) from the possible swap candidates, since the values of vectors 1 and 3 are required for the immediate computation. For this reason, `getxVector()` has two additional parameters that specify which inner nodes must be pinned to memory and can not be swapped out.

Even if we optimize data transfer performance between disk and RAM at a technical level, accessing data on disk has a significantly higher latency than accessing data in RAM. Therefore, it is important to minimize the number of I/O accesses (number of ancestral probability vector swaps).

As already mentioned, a vector is always either stored on disk or in RAM. Whenever RAxML tries to access a vector that resides on disk via `getxVector()`, we need to chose a vector that resides in RAM, and then swap the vectors. Therefore, we require a replacement strategy, that is conceptually similar to cache line replacement or page swap strategies.

To conduct a thorough assessment, we have implemented and tested the following four replacement strategies:

Random The vector to be replaced is chosen at random with minimum overhead (one call to a random number generator).

LRU Least Recently Used. The vector to be replaced is the one that has been accessed the furthest back in time. This requires a list of n time-stamps as well as an $O(log(n))$ binary search for the oldest time-stamp. Note the use of n rather than m, because we only search among time stamps of vectors that are currently in RAM.

LFU Least Frequently Used. The vector to be replaced is the one which has been accessed the least number of times. This requires maintaining a list of m entries containing the access frequency and an $O(log(n))$ binary search for the smallest value.

Topological The vector to be replaced is the most distant node (in terms of number of nodes along the path in the tree) from the node/vector currently being requested. The node distance between a pair of nodes in a binary tree is defined as the number of nodes along the unique path that connects them.

The rationale for the topological replacement strategy is that, due to the locality of the tree search and the computations, we expect the most distant node/vector to be accessed again the furthest ahead in the future.

Reducing the Number of Swaps

So far, our EM algorithm has been integrated into RAxML in a fully transparent way. We have shown that it is possible to modify the program and any PLF-based program for that matter, such that the complexity is entirely encapsulated by a function call that returns the address of an ancestral probability vector. However, it is possible to further reduce the number of I/O operations by exploring some implementation-specific characteristics of RAxML, that can also be deployed analogously in other PLF-based implementations.

For each global or local traversal (re-computation of a part or of all ancestral probability vectors) of the tree, we know, a priori (based on the tree structure), that some of the vectors that will be swapped into RAM will be completely overwritten, that is, they will be used in write-only mode during the first access. Thus, whenever we swap in a vector from file, of which

we know that it will initially be used for writing, we can omit reading its current contents from file. We denote this technique as *read skipping* and implement it as follows: We introduce a flag in our EM bookkeeping data structure that indicates whether read skipping can be applied or not, that is whether a vector will be written during the first access. We instrument the search algorithm such that, when the global or local tree traversal order is determined (this is done prior to the actual likelihood computations), the flag is set appropriately.

3.2.3 Experimental Setup & Results

Evaluation of replacement strategies

Assessing the correctness of our implementation is straight-forward since this only requires comparing the log likelihood scores obtained from tree searches using the standard RAxML version and the out-of-core version. Hence, we initially focus on analyzing the performance (vector miss rate) of our replacement strategies and the impact of the *read skipping* technique as a function of f (proportion of vectors residing in RAM).

To evaluate the replacement strategies, we used 2 real-world biological datasets with 1288 species (DNA data, MSA length $s := 1200$ sites/columns), and 1908 species (DNA data, MSA length: $s := 1424$ sites/columns) respectively. Tree searches were executed under the Γ model of rate heterogeneity with four discrete rates. We used the SSE3-based [7] sequential version of RAxML v7.2.8.

For each of the four replacement strategies, we performed three different runs for the out-of-core version with $f := 0.25$ (25% of vectors memory-mapped), $f := 0.50$ (50% of vectors memory-mapped), and $f := 0.75$ (75% of vectors memory-mapped).

Given a fixed starting tree, RAxML is deterministic, that is, regardless of f and the selected replacement strategy, the resulting tree (and log likelihood score) must always be identical to the tree returned by the standard RAxML implementation. For each run, we verified that the standard version and the out-of-core version produced exactly the same results.

This part of our computational experiments was conducted on a single core of an unloaded multi-core system (Intel Xeon E5540 CPU running at 2.53GHz with 36 GB of RAM). On this system, the amount of available RAM was sufficient to hold all vectors in memory for the two test-datasets, both for the standard implementation or by using memory-mapped I/O for the out-of-core version.

We found that, with the exception of the LFU strategy, even mapping

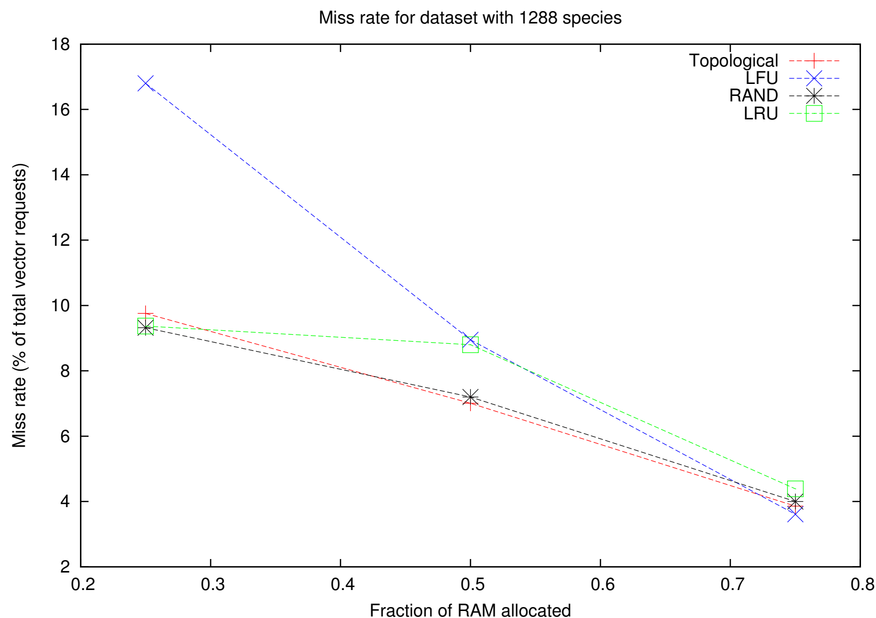

Figure 3.3: Vector miss rates for different replacement strategies using a dataset with 1,288 species. We allocated 25%, 50% and 75% of the required memory for storing ancestral probability vectors in RAM. Every time a vector is not available in RAM, we count a miss.

only 25% of the probability vectors to memory results in miss rates under 10%. As expected, miss rates converge to zero as the fraction of available RAM is increased (see Figure 3.3). In the trivial case ($f := 1.0$), the miss rate is zero, since all vectors reside in RAM. The Random, LRU, and Topological strategies perform almost equally well. Thus, one would prefer the random or LRU strategy over the topological strategy because it requires a larger computational overhead for determining the replacement candidate.

In Figure 3.4, we show the positive effect of the *read skipping* technique for analogous runs on the same dataset with 1288 species. Here, we quantify the fraction of ancestral vector reads from disk that are actually carried out per ancestral vector access. Note that, without the *read skipping* technique this fraction would be identical to the miss rate in Figure 3.3. Thus, by deploying this technique, we can omit more than 50% of all vector read operations.

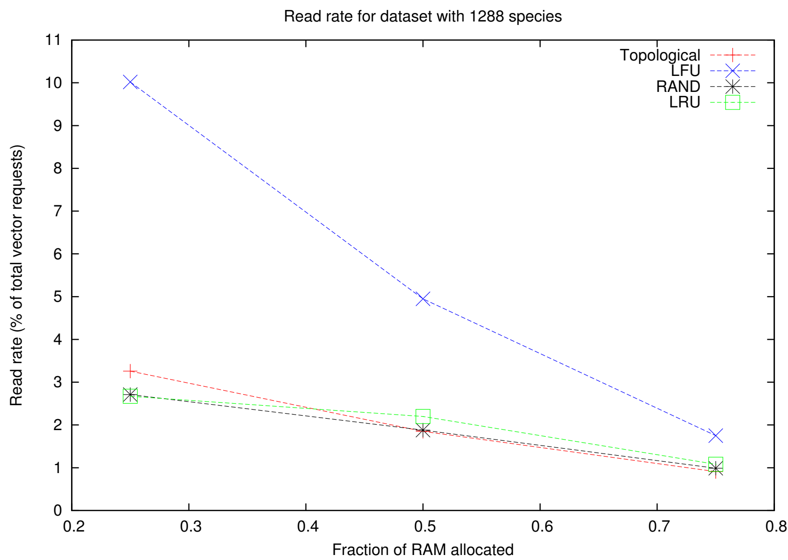

Figure 3.4: Effect of Read skipping: We count the fraction of vector accesses for which a vector needs to be actually read from file using the same parameters as in Figure 3.3. Without the read skipping strategy the read rate is equivalent to the miss rate.

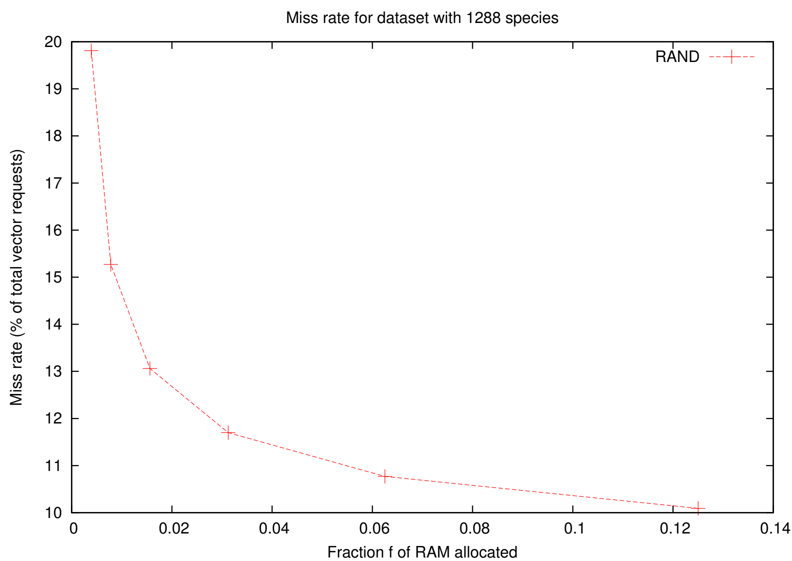

Figure 3.5: Miss rates for several runs using the random replacement strategy on the dataset with 1288 species. The fraction f of memory-mapped ancestral probability vectors was divided by two for each run.

Miss Rates as a Function of f

To more thoroughly assess the impact of f on the miss rate, we conducted additional experiments for different values of f using a random replacement strategy on the test dataset with 1288 taxa. The fraction f was subsequently divided by two. The smallest f value we tested corresponds to only five ancestral probability vector slots in RAM.

Figure 3.5 depicts the increase in miss rates for decreasing f. The most extreme case with only five RAM slots, still exhibits a comparatively low miss rate of 20%. This is due to the good vector usage locality in the RAxML algorithm. One reason for this behavior, that is inherent to ML programs, are branch length optimization procedures. Branch length optimization is typically implemented via a Newton-Raphson procedure, that iterates over a single branch of the tree. Thus, only memory accesses to the same two vectors (located at either end of the branch) are required in this phase which accounts for approximately 20-30% of overall execution time. In RAxML, access locality is also achieved by —in most cases— only re-optimizing three branch lengths after a change of the tree topology during the tree search (Lazy SPR technique; see [87]).

Real Test Case

Finally, we conducted realistic tests by analyzing large data sets on a system with only 2GB of RAM. Here, we compare execution times of the standard algorithm (using paging) with the out-of-core performance.

For these tests, we generated large simulated DNA datasets using INDELible [26]. We intentionally generated datasets whose memory requirements for storing ancestral probability vectors (see Figure 3.6) exceed the main memory available on the test system (Intel i5 running at 2.53 GHz with 2GB RAM configured with 36 GB swap space). To achieve this, we deployed INDELible to simulate DNA data on a tree with 8192 species and varying alignment lengths s. We chose values of s such that, the simulated datasets had (ancestral probability) memory requirements ranging between 1GB and 32GB. Because of the prohibitive execution times for full tree searches on such large datasets, we did not execute the standard RAxML search algorithm. Instead, we executed a subset of the PLF as implemented in RAxML (`-f z` option in the modified code by simply reading in a given, fixed, tree topology and computing five full tree traversals (recomputing all ancestral probability vectors in the tree five times) according to the Felsenstein pruning algorithm. This represents a worst-case analysis, since full tree traversals exhibit the smallest degree of vector locality. Full tree traversals are required

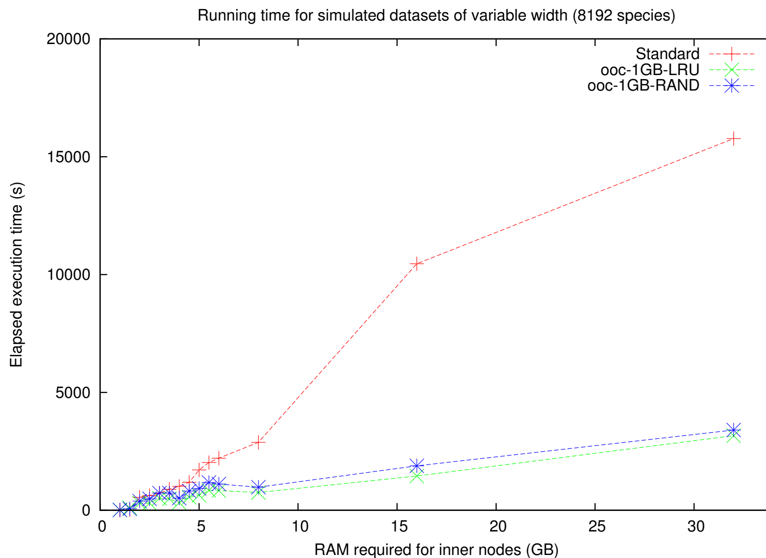

Figure 3.6: Execution times for 5 full tree traversals on a tree with 8192 sequences and variable dataset width s for the standard RAxML version (using paging) and the out-of-core version.

to optimize likelihood model parameters such as the α shape parameter of the Γ model of rate heterogeneity.

The out-of-core runs were invoked with the `-L 1,000,000,000` flag to force the program to use less than 1GB of RAM for storing ancestral probability vectors.

Figure 3.6 demonstrates that the execution times of the out-of-core implementation scale well with dataset size (ancestral probability vector space requirements). As expected, the standard approach is faster for datasets that still fit into RAM, while it is more than five times slower for the largest dataset with 32GB. For ancestral vector memory footprints between 2GB and 32GB, the run-time performance gap between the out-of-core implementation and the standard version increases, since the standard algorithm starts using virtual memory (e.g., then number of page faults increases from 346,861 for 2GB to 902,489 for 5GB). Note that, for the out-of-core runs, we only use 1GB of RAM, that is, better performance can be achieved by slightly increasing the value for `-L`. Thus, under memory limitations and given the runtimes in Figure 3.6 using the out-of-core approach is significantly faster than the standard approach for computing the likelihood on large datasets.

3.3 Recomputation of Ancestral Vectors

The content of this Section has been derived from the following peer-reviewed publication:
F. Izquierdo-Carrasco, J. Gagneur, and A. Stamatakis. Trading running time for memory in phylogenetic likelihood computations. In J. Schier, C. M. B. A. Correia, A. L. N. Fred, and H. Gamboa, editors, *BIOINFORMATICS*, pages 86–95. SciTePress, 2012
J. Gagneur is the author of the proof in Figure 3.3.1 on the theoretical minimum memory requirements to compute the PLF.

Given enough execution time and disk space, the out-of-core version can be deployed to essentially infer trees on datasets of arbitrary size. However, the increase in execution times in the out-of-core implementation is still large. While we are confident that the miss rate and miss penalty can be further improved by low-level I/O performance optimization, we have explored the overhead of, instead of storing on disk, recomputing ancestral probability vectors.

In this Section we introduce the implementation of a versatile concept to trade memory for additional computations in the likelihood function which exhibits a surprisingly small impact on overall execution times. When trading 50% of the required RAM for additional computations, the average execution time increase because of additional computations amounts to only 15%. This slowdown (due to the additional recomputations) is comparable to the standard deviation of the running time of a set of independent searches, that is, it is not significant from the user perspective. We demonstrate that, theoretically, for a phylogeny with n species only $log(n) + 2$ memory space is required for computing the likelihood of a tree. We also assess practical lower bounds.

The term time-memory trade-off engineering refers to situations/approaches where memory requirements/utilization is reduced at the cost of additional computations, that is, at the expense of increased run time. This trade-off engineering approach has been applied in diverse fields such as language recognition [18], cryptography [5], and packet scheduling [109]. We are not aware of any applications of or experiments with time-memory trade-off engineering approaches in the field of computational phylogenetics.

We next describe the underlying idea for the memory saving approach, demonstrate that only $log(n) + 2$ vectors are required to compute the PLF,

and introduce two vector replacement strategies. In Subsection 3.3.2 we describe the experimental setup and provide corresponding results.

3.3.1 Recomputation of Ancestral Probability Vectors

Specific PLF Memory Requirements

We have previously discussed the memory requirements for the PLF computation in Section 3.1. Next, we will discuss how these requirements can be lowered with the recomputation trade-off. Let us consider again Equation 2.10 for the PLF computation of one entry in the ancestral probability vector of node p at site i.

$$L_A^{(p)}(i) = \Big(\sum_{S=A}^{T} P_{AS}(b_{lp}) L_S^{(l)}(i)\Big)\Big(\sum_{S=A}^{T} P_{AS}(b_{rp}) L_S^{(r)}(i)\Big)$$

In order to compute $L_A^{(p)}(i)$, we do not necessarily need to immediately have available in memory vectors $L_S^{(l)}(i)$ and $L_S^{(r)}(i)$, since they can be obtained by a recursive descent (a post-order subtree traversal) into the subtrees rooted at l and r using the above equation. In other words, if we do not have enough memory available, we can recursively re-compute the values of $L^{(l)}$ and $L^{(r)}$. Note that, the recursion terminates when we reach the tips (leaves) of the tree and that memory requirements for storing tip vectors are negligible compared to ancestral vectors.

If not all ancestral probability vectors fit in RAM, the required vectors for the operation at hand can still be obtained by conducting some additional computations for obtaining them by applying equation 2.10. We observe that, when both $L_S^{(l)}(i)$ and $L_S^{(r)}(i)$ have been used (consumed) for calculating $L_A^{(p)}(i)$, the two child vectors are not required any more. That is, those two vectors can be omitted/dropped from RAM or be overwritten in RAM to save space. Therefore, the likelihood of a tree can be computed without storing all $n-2$ ancestral vectors. Instead, a smaller amount of space for only storing $x < (n-2)$ vectors can be used. Those x vectors can then dynamically be assigned to a subset (varying over time) of the $n-2$ inner nodes. This gives rise to the following question: Given a MSA with n taxa, what is the minimum number of inner vectors x_{min} that must reside in memory to be able to compute the likelihood on any unrooted binary tree topology with n taxa?

Due to the post-order traversal of the binary tree topology, some ancestral vectors need to be stored as intermediate results. Consider an ancestral node p that has two subtrees rooted at child nodes l and r with identical subtree

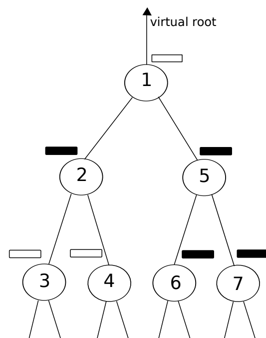

Figure 3.7: A balanced subtree, where the vector of inner node 1 is oriented in the direction of the virtual root. In order to compute the likelihood of vector 1, the bold vectors must be held in memory. The transparent vectors may reside in memory but are not required to.

depth. When the computations on the first subtree for obtaining $L^{(l)}$ have been completed, this vector needs to be kept in memory until the ancestral probability vector $L^{(r)}$ has been calculated to be able to compute $L^{(p)}$.

In the following Figure 3.3.1, we demonstrate that the minimum number of ancestral vectors $x_{min}(n)$ that must reside in RAM to be able to compute the likelihood of a tree with n taxa is $log_2(n) + 2$. For obtaining this lower memory bound, we consider the worst case tree shape, that is, a perfectly balanced unrooted binary tree with the virtual root placed into the innermost branch. As we show in Figure 3.3.1, this represents the worst case because all subtrees from the virtual root have the same length.

Figure 3.7, where we first descended into the left subtree, depicts this worst case scenario for $n = 4$. The probability vectors will be written in the following order: `3, 4, 2, 6, 7, 5, 1`. Figure 3.7 shows the number of vectors required in memory ($log_2(4) + 2 = 4$) at the point of time where vector 5 needs to be computed. At any other point of time during the post-order traversal, holding 3 vectors in RAM is sufficient to successfully proceed with the computations.

Theoretical Minimum Memory Requirements

The underlying idea of our approach is to reduce the total number of ancestral probability vectors residing in RAM at any given point of time. Let PLF be the likelihood function (see Equation 2.10), which —for the sake of simplifying our notation— can be computed at a tip (essentially at zero computational cost) or an inner node given the ancestral probability vectors of its two child nodes. The PLF always returns an ancestral probability vector

as result.

As a pre-processing step, we compute the number of descendants (size of the respective subtree) for each inner node. This can be implemented via a single post-order tree traversal. The memory required for storing the number of descendants as integers at each node is negligible compared to required probability vector space. Given this information, the binary tree data structure can be re-ordered such that, for every node p, the left child node l contains the larger number of descendants and the right child node r contains the smaller number of descendants. We can now compute the PLF of the tree by invoking the following recursive procedure **f** at the virtual root p of the tree:

```
proc f(p) ≡
  if isALeaf(p) then return(PLF(p)) fi;
  v_l := f(p.l);                                Step 1
  v_r := f(p.r);                                Step 2
  return(PLF(v_l, v_r))                         Step 3
.
```

During this computation, the maximum number of probability vectors $x_{min}(n)$ that need to reside in memory for any tree with n leaves is $\log_2(n)+2$. This upper bound is required *if and only if* the binary tree is balanced.

Proof: We demonstrate the above theorem by recursion. $x_{min}(1) = 1$, since for a tree with a single node, only the sequence data (a single probability vector) need to be stored. Now assume that $x_{min}(n-1) = \log_2(n-1) + 2$ and consider a tree with n leaves. We execute all steps of $f(p)$, where p is the root, and keep track of the number of probability vectors that are stored simultaneously in RAM:

- **Step 1:** Computing $f(p.l)$ requires storing at most $x_{min}(n-1)$ probability vectors simultaneously which is strictly less than $\log_2(n) + 2$. Once $f(p.l)$ has been computed only the result vector is stored (i.e., only one single probability vector).

- **Step 2:** Because the subtree size of the right descendant of the root $p.r$ must be $\leq n/2$, this computation needs to simultaneously store at most $\log_2(n/2) + 2 = \log_2(n) + 1$ probability vectors. Here, we need to add the number of probability vectors that are required to be maintained in RAM after completion of **Step 1**. Thus, overall, **Step 2** needs to simultaneously hold at most $\log_2(n) + 2$ probability vectors in RAM.

Once the results of $f(p.l)$ and $f(p.r)$ have been computed we are left with two probability vectors v_l, v_r that need to reside in memory.

- **Step 3:** This step requires 3 probability vectors to be stored at the same time, namely, v_l, v_r, and the vector to store the result of $PLF(v_l, v_r)$. Note that, to obtain the overall likelihood of the tree (a single numerical value), we need to apply a function $g()$ to the single result vector returned by $f()$, that is, $g(f(p))$. Function $g()$ simply uses the data in the result vector to calculate the likelihood score over all root vector entries.

Hence the peak in memory usage is reached during **Step 2** and its upper bound is $\log_2(n)+2$. Moreover, this upper bound is reached *if and only if* the number of descendants in the respective left and right subtrees is identical for all nodes, that is, for a balanced tree.

Basic Implementation

While holding $log_2(n) + 2$ vectors in RAM provides sufficient space for computing the PLF, this will induce a significant run-time increase (due to recomputations) compared to holding n vectors in memory. In practice, we need to analyze and determine a reasonable run-time versus RAM tradeoff as well as an appropriate vector replacement/overwriting strategy. For instance, in the RAxML or MrBayes PLF implementations, some ancestral vectors (depending on their location in the tree) can be recomputed faster than others. In particular, cases where the left and/or right child vectors are tip sequences can be handled more efficiently. For instance, an observed nucleotide `A` at a tip sequence corresponds to a simple probability vector of the form $[P(A) := 1.0, P(C) := 0.0, P(G) := 0.0, P(T) := 0.0]$. This property of tip vectors can be used for saving computations in equation 2.10.

Devising an appropriate strategy (see Section 3.3.1) for deciding which vectors shall remain in RAM and which can be discarded (because they can be recomputed at a lower computational cost) can have a substantial impact on the induced run time overhead when holding, for instance, $x := n/2$ vectors in RAM. In the following, we will outline how to compute the PLF and conduct SPR-based tree searches with $x < n$ vectors in RAM.

Let n be the number of species, $n-2$ the number of ancestral probability vectors, and x the number of available slots in memory, where $log_2(n) + 2 \leq x < n$ (i.e., $n-x$ vectors are not stored, but recomputed on demand). Let w be the number of bytes required for storing an ancestral probability vector (all vectors have the same size). Our implementation will only allocate $x \cdot w$

bytes, rather than $n \cdot w$. We henceforth use the term *slot* to denote a RAM segment of w bytes that can hold an ancestral probability vector.

We define the following C data structure (details omitted) to keep track of the vector-to-slot mapping of all ancestral vectors and for implementing replacement strategies:

```
typedef struct
{
  int numVectors;
  size_t width;
  double **tmpvectors;
  int *iVector;
  int *iNode;
  int *unpinnable;
  boolean allSlotsBusy;
  unpin_strategy strategy;
}recompVectors;
```

The array `tmpvectors` is a list of pointers to a set of slots (i.e., starting addresses of allocated RAM memory) of size `numVectors` (x). The array `iVector` has length x and is indexed by the slot id. Each entry holds the node id of the ancestral vector that is currently stored in the indexed slot. If the slot is free, the value is set to a dedicated `SLOT_UNUSED` code. The array `iNode` has length $n - 2$ and is indexed using the unique node ids of all ancestral vectors in the tree. When the corresponding vector resides in RAM, its array entry holds the corresponding slot id. If the vector does not reside in RAM the array entry is set to the special code `NODE_UNPINNED`. Henceforth, we denote the availability/unavailability of an ancestral vector in RAM as `pinned/unpinned`. The array `unpinnable` tracks which slots are available for unpinning, that is, which slots that currently hold an ancestral vector can be overwritten, if required.

The set of ancestral vectors that are stored in the memory slots changes dynamically during the computation of the PLF (i.e., during full tree traversals and tree searches). The pattern of dynamic change in the slot vector also depends on the selected recomputation/replacement strategy. For each PLF invocation, be it for evaluating a SPR move or completely re-traversing the tree, the above data structure is updated accordingly to ensure consistency.

Whenever we need to compute a local tree traversal (following the application of an SPR move) to compute the likelihood of the altered tree topology, we initially just compute the traversal order which is part of the standard RAxML implementation. The traversal order is essentially a list that stores in which order ancestral probability vectors need to be computed. In other

words, the traversal descriptor describes the partial or full post-order tree traversal required to correctly compute the likelihood of a tree. For using $x < n$ vectors, we introduce a so-called traversal order check, which extends the traversal steps (the traversal list) that assume that all n vectors reside in RAM. By this traversal order extension, we guarantee that all missing vectors (not residing in RAM) will be recomputed as needed. The effect of reducing the number of vectors residing in RAM is that, traversal lists become longer, that is, more nodes are visited and thereby run time increases. When the traversal is initiated, all vectors in the traversal list that already reside in RAM (they are pinned to a slot) are protected (marked as `unpinnable`) such that, they will not be overwritten by intermediate vectors of the recomputation steps.

If an ancestral vector slot needs to be written/stored by the traversal, there are three cases:

1. The vector resides in a slot (already in memory). We can just read and/or write to this slot.

2. The vector is not pinned, but there exists a free slot, which is then pinned to this vector.

3. The vector is not pinned and there is no free slot available. A residing vector must be replaced and the corresponding slot needs to be pinned to the required vector.

Since the traversal only visits each vector at most once, the corresponding children of this vector can be unpinned once it has been written to memory. Instead of unpinning them directly, they are merely marked for unpinning. The real overwrite only takes place if the slot is selected by the replacement strategy for storing another vector. Otherwise, the slot will store the values of the current vector for as long as possible for potential future re-use.

Replacement Strategies

In analogy to the replacement strategies discussed in the out-of-core implementation in Section 3.2, there are numerous approaches for deciding which memory slot should be overwritten by a new ancestral vector that does currently not reside in RAM. We implement and analyze the following two replacement strategies.

Random A random slot not flagged as pinned is selected. The random strategy is a naïve approach with minimal overhead and is used as baseline for performance comparisons.

MRC Minimum Recomputation Cost. The slot with minimum subtree size (see below) and not flagged as pinned is selected.

The MRC strategy entails a slight overhead for keeping track of which vectors will be most expensive to recompute and should therefore be kept in RAM for as long as possible. Consider an unrooted binary tree T with n tips. Each inner node i in an unrooted binary tree with n taxa can be regarded as a trifurcation that defines three subtrees $T_{i,a}$, $T_{i,b}$, and $T_{i,c}$ corresponding to the three outgoing branches a, b, and c. Given a subtree $T_{i,x}$, we define its subtree size $sts(T_{i,x})$ as the number of tips beneath inner node i in the direction of the outgoing branch x. Thus, $sts(T_{i,a}) + sts(T_{i,b}) + sts(T_{i,c}) = n$ holds for any inner node i in an unrooted binary tree with n taxa/tips. When conducting likelihood computations, the tree is always rooted by a virtual root. Hence, if the virtual root is located in the direction of branch c, the relevant subtree size with respect to the recomputation cost at an inner node i is $sts(T_i^{rooted}) := sts(T_{i,a}) + sts(T_{i,b})$. We use this rooted subtree size $sts(T_i^{rooted})$ to determine the recomputation cost for each ancestral vector i, given a virtual rooting of the tree. In particular, the case $sts(T_i^{rooted}) = 2$ (both children are tips) is particularly cheap to recompute, since tip vectors always reside in RAM *and* recomputing ancestral vector i is cheap (see above). In a perfectly balanced tree with the root placed in the innermost branch, half of the inner vectors have subtree size $sts(T_i^{rooted}) = 2$.

As already mentioned, during a partial or full tree traversal for computing the likelihood, all inner nodes (vectors) involved are oriented in a given direction toward the virtual root. Evidently, the subtree sizes will change when the topology is altered and will need to be updated accordingly.

In order to account for this, we keep an array of subtree sizes, that is, for each inner node we store a subtree size value. Whenever the topology changes, a local traversal descriptor is created. This local traversal descriptor starts at the new virtual root and recursively includes the inner nodes whose orientation has changed after the given topology change. Since this exactly corresponds to the set of nodes whose subtree size values must be updated, the subtree size array can be updated via the same recursive descent procedure.

Implementation Details

In the following we discuss some important details of the recomputation process.

Largest subtree first The standard implementation of the PLF, where all ancestral vectors are available in memory, computes the PLF by conducting

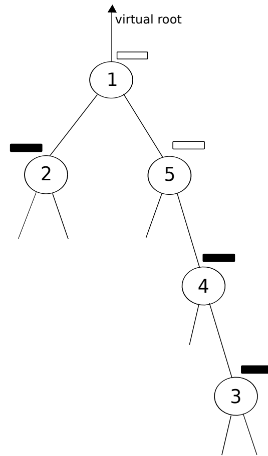

Figure 3.8: An unbalanced subtree, where the vector of inner node 1 is oriented in the direction of the virtual root. Bold rectangles represent vectors that must be held in memory if we first descend into the left subtree and node 4 is being written.

a post-order traversal from an arbitrary rooting of the tree. Thus, an ancestral probability vector can be computed once the respective left and right child vectors have been computed. In the standard RAxML implementation, the traversal always recursively descends into the left subtree first. The (arbitrary) choice whether to descend into the left or right subtree first, does not affect performance (nor the results) when all ancestral vectors reside in RAM.

However, when not all vectors reside in RAM, the choice whether to descend into the left or right subtree first *does* matter. This is particularly important if we use the minimum setting $x := log_2(n) + 2$, since otherwise we may encounter situations where not enough slots are available (see Figure 3.3.1).

Suppose that, as in the standard implementation, we always descend into the left subtree first. In the example shown in Figure 3.8, the left subtree is significantly smaller than the right subtree. We would first descend into the left subtree, which consists of a single inner node. The child ancestral vector corresponding to node 2 must be pinned to its slot. Thereafter, we descend into the right subtree writing and pinning nodes 3, 4, 5 (always assuming that we descend into the left —smaller— subtree of *each* node first). While we keep descending into the right subtree, the ancestral vector corresponding to node 2 remains pinned, because it represents an intermediate result.

In contrast, if we initially descend into the right subtree (which is always

larger in the example), there is no need to store intermediate results of the left subtree (node 2). Also, nodes 4 *and* 5 can be immediately unpinned as soon as their parent nodes have been computed. Thus, by inverting the descent order such as to always descend into the larger subtree first (as required by our proof), we minimize the amount of time for which intermediate vectors must remain pinned. Note that, when two subtrees have the same size, it does not matter into which subtree we descend first.

If we descend into the smaller subtrees first, there will be more vectors that need to remain pinned for a longer time. This would also reduce the effective size of the set of inexpensive-to-recompute vectors that shall preferentially be overwritten, because more vectors that are cheap to recompute need to remain pinned since they are holding intermediate results. In this scenario more expensive-to-recompute vectors will need to be overwritten in memory and dropped from RAM.

Determining the appropriate descent order (largest subtree first) is trivial and induces a low computational overhead. When the traversal list is computed, we simply need to compare the subtree sizes of child nodes and make sure to always descend into the largest subtree first.

Priority List For this additional optimization, we exploit a property of the SPR move technique. When a candidate subtree is pruned (see Figure 2.9) from the currently best tree, it will be re-inserted into several branches of the tree from which it was removed to evaluate the likelihood of different placements of the candidate subtree within this tree.

In the course of those insertions, the subtree itself will not be changed and only the ancestral vector at the root of the subtree will need to be accessed for computations. Hence, we maintain a dedicated list of pruned candidate subtree nodes (a unpinning priority list) that can be preferentially unpinned. Because of the design of lazy SPR moves in RAxML (similar SPR flavors are used in GARLI and PHYML) those nodes (corresponding to ancestral vectors) will not be accessed while the candidate subtree is inserted into different branches of the tree. Once this priority list is exhausted, the standard MRC recomputation strategy is applied.

Full Traversals Full post-order traversals of the tree are required during certain phases of typical phylogenetic inference programs, for instance when optimizing global maximum likelihood model parameters (e.g., the α shape parameter of the Γ distribution or the rates in a GTR matrix) on the entire tree. Full tree traversals are also important for the post-analysis of fixed tree topologies, for instance, to estimate species divergence times. Full tree

traversals represent a particular case because every inner vector of the tree needs to be visited and computed. Hence, the number of vectors that need to be computed under our memory reduction approach is exactly identical to the number of vectors that need to be computed under the standard implementation. Thus, there is no need for additional computations while a large amount of memory can be saved. While full tree traversals do not dominate run times in standard tree search algorithms, they can dominate execution times in other phylogenetic downstream analysis, such as, e.g., branch length optimization or estimation of species divergence times and lineage-specific substitution rates [31].

3.3.2 Experimental Setup and Results

We have implemented the techniques described in Subsection 3.3.1 in RAxML-Light v1.0.4 [91]. RAxML-Light is a strapped-down dedicated version of RAxML intended for large-scale phylogenetic inferences on supercomputers. It implements a light-weight software-based checkpointing mechanism and offers fine-grained PThreads and MPI parallelizations of the PLF. It has been used to compute a tree on a dense simulated MSA with 1481 taxa and 20,000,000 sites that required 1TB of RAM and ran in parallel with MPI on 672 cores. RAxML-Light will not be maintained anymore, but the ExaML code [90], which uses a more efficient MPI paralellization strategy, can be used for phylogenetic inferences on supercomputers. The recomputation implementation used for this evaluation is a fork of RAxML-Light, and available at `https://github.com/fizquierdo/vectorRecomp`

Evaluation of recomputation strategies

The recomputation algorithm yields *exactly* the same log likelihood scores as the standard algorithm. Thus, for validating the correctness of our implementation, it is sufficient to verify that the resulting trees and log likelihood scores of a ML tree search with the standard and recomputation implementations are identical. The increase of total run time depends on the number x of inner vectors that are held in memory *and* on the chosen unpinning strategy (MRC versus RANDOM). We used INDELible [26] to generate simulated MSAs of 1500, 3000, and 5000 species. All experiments described in this Section were conducted for these three datasets. For this purely computational work it does not matter whether simulated or real data are used.

Initially, we ran the `Parsimonator` program [85] to generate 10 distinct randomized stepwise addition order parsimony starting trees for each MSA. For each starting tree, we then executed a standard ML tree search with

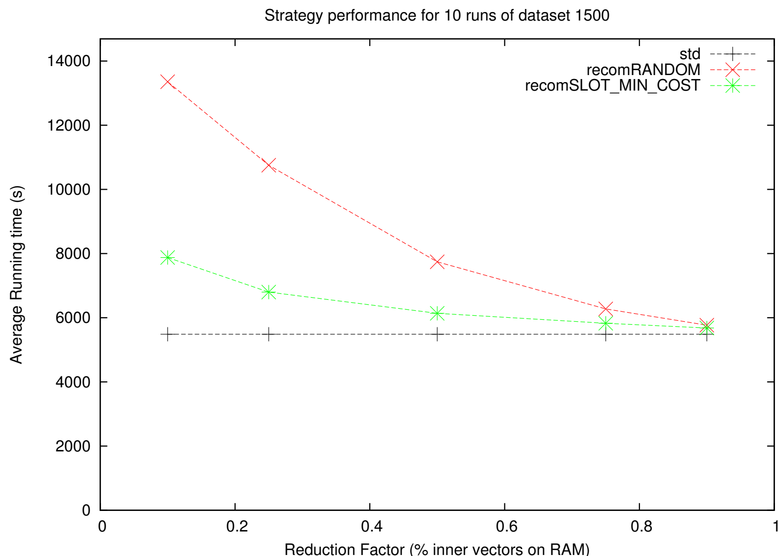

Figure 3.9: Different replacement strategies. The dataset was run with RAM allocations of 10%, 25%, 50% , 75%, and 90%, of the total required memory for storing all probability vectors. Run times are averaged across 10 searches with distinct starting trees.

RAxML-Light (sequential version 1.0.4 with SSE3 intrinsics) and ML tree searches with the recomputation version for the two replacement strategies (MRC and RANDOM) and five different RAM reduction factors (`-x` and `-r` options respectively). In the recomputation version used for the evaluation, both the *Largest subtree first* and the *Priority List* optimizations were activated. All experiments were executed on a 48-core AMD system with 256GB RAM. For all runs, RAM memory usage was measured every 600 seconds with `top`.

Figure 3.10 shows the corresponding decrease in RAM usage. Values in Figure 3.10 correspond to maximum observed RAM usage values.

Figure 3.9 depicts the run time increase as a function of available space for storing ancestral probability vectors.

Clearly, the MRC strategy outperforms the RANDOM strategy and the induced run-time overhead, even for a reduction of available RAM space to only 10% is surprisingly small (approximately 40%). This slowdown is acceptable, considering that instead of analyzing a large dataset on a machine with 256GB, a significantly smaller and less expensive system with, for instance, 32GB RAM will be sufficient.

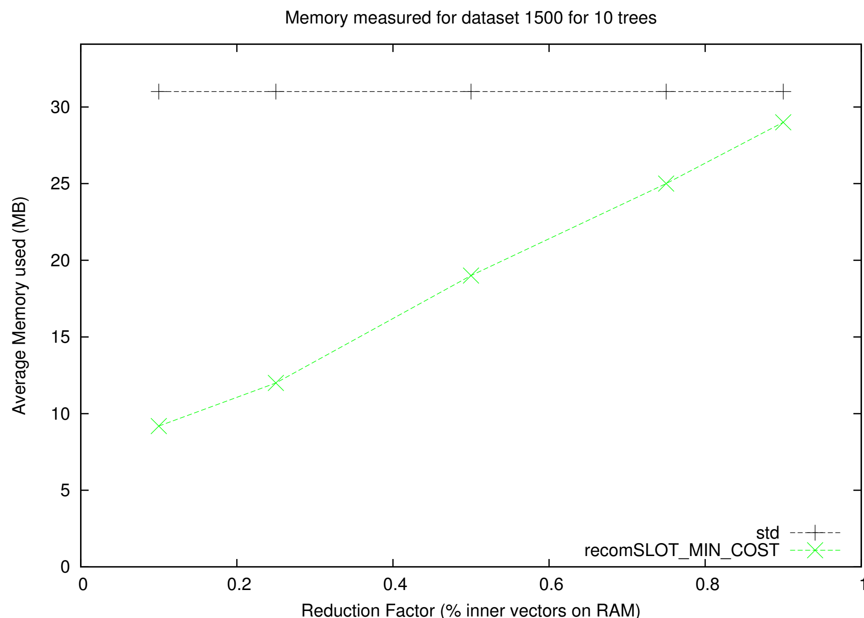

Figure 3.10: Overall RAM usage when allocating only 10%, 25%, 50% , 75%, and 90%, of the required ancestral probability vectors.

3.3.3 Evaluation of traversal overhead

In order to evaluate the overhead of the extended (larger) tree traversals due to the required additional ancestral probability vector computations, we modified the source code to count the number of ancestral vector computations. We distinguish between three cases with different recomputation costs (see Subsection 3.3.1). For each case, there exists a dedicated PLF implementation in RAxML.

Tip/Tip Both child nodes are tips.

Tip/Vector One child node is a tip, and the other is an ancestral vector (subtree).

Vector/Vector Both child nodes are ancestral vectors.

Table 3.1 shows a dramatic, yet desired increase, in the number of Tip/Tip vector computations for the MRC strategy. The amount of the slowest type of ancestral node computations [Vector/Vector], however, is only increased by 0.16% compared to the standard implementation.

Strategy	Tip/Tip	Tip/Vector	Vector/Vector	Total	Runtime (s)
Standard	11,443,484	57,884,490	76,325,233	145,653,207	5678
MRC (0.5)	20,368,957	61,224,562	76,444,874	158,038,393	6453
Random (0.5)	37,778,575	85,303,730	104,398,910	227,481,215	7999

Table 3.1: Frequency of ancestral vector cases for the standard implementation and the recomputation strategies (50% of ancestral vectors allocated)

Dataset	Standard	R:=0.1	R:=0.9
500	0.122	0.121	0.130
1500	0.430	0.430	0.434
5000	2.402	2.412	2.438

Table 3.2: Average run times in seconds for 20 full traversals averaged across 5 runs

Evaluation of full tree traversals

We created a simple test case that parses an input tree and conducts 20 full tree traversals on the given tree. We used the aforementioned starting trees and the 500, 1500, and 5000 taxon datasets. Each run was repeated 5 times and we averaged running times. All runs returned exactly the same likelihood scores.

Table 3.2 indicates that, even for very small R values (fraction of inner vectors allocated in memory), the run time overhead is negligible compared to the standard implementation.

The recomputation approach permits to save memory at the cost of some additional computations, and independently of the data pattern in the alignment. However, certain data patterns in the MSAs can be exploited to save memory without any overhead in computations. In fact, as we show in the next Section, if large blocks of gaps are present in the alignment, it is possible to reduce both the number of computations and the memory requirements.

3.4 Subtree Equality Vectors

3.4.1 Gappy Subtree Equality Vectors

The content of this Section has been derived from the following peer-reviewed publication:
F. Izquierdo-Carrasco, S. Smith, and A. Stamatakis. Algorithms, data structures, and numerics for likelihood-based phylogenetic inference of huge trees. *BMC Bioinformatics*, 12(1):470, 2011
S. Smith generated the 38K and 56K datasets described in Subsection 3.4.2

The concept of Subtree Equality Vectors (SEVs) to accelerate likelihood computations by reducing the number of required floating point operations was first introduced in 2002 [96]. Conceptually similar approaches were presented in 2004 [64] and 2010 [100].

The underlying idea is based on the following observation: Given two identical alignment sites i and j in the MSA that evolve under the same evolutionary model (GTR parameters, α shape parameter of the Γ function, etc.), and for which a joint branch length has been estimated, their per-site log likelihoods $LnL(i)$ and $LnL(j)$ will be identical. Hence, to save computations, one can compress the identical sites into a single site pattern and assign a respective site pattern count (weight) to this site pattern. Thus, for two identical sites i and j, we can compute the per-site log likelihood as $2 \cdot LnL(i)$. This global compression of alignments (executed prior to conducting likelihood computations) is implemented in all current likelihood-based codes. This basic idea of site compression can be extended to the subtree level, by using SEVs for instance, to save additional computations. Let us consider again Equation 2.10, which computes the ancestral probability entry to observe state A at node p and site i.

$$L_A^{(p)}(i) = \Big(\sum_{S=A}^{T} P_{AS}(b_{lp}) L_S^{(l)}(i)\Big)\Big(\sum_{S=A}^{T} P_{AS}(b_{rp}) L_S^{(r)}(i)\Big) \tag{2.10}$$

We can now consider again Equation 3.1, which computes the ancestral probability entry to observe state A at node p and site j.

$$L_A^{(p)}(j) = \Big(\sum_{S=A}^{T} P_{AS}(b_{lp}) L_S^{(l)}(j)\Big)\Big(\sum_{S=A}^{T} P_{AS}(b_{rp}) L_S^{(r)}(j)\Big) \tag{3.1}$$

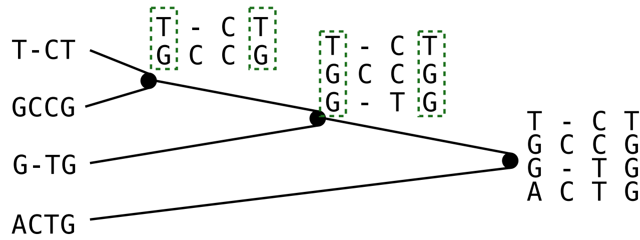

Figure 3.11: All alignment columns are different. Therefore, at the root, each site will contribute with a different site likelihood. However, at the internal nodes, the first and the last site share identical patterns (green) at the node level sub-alignment. Since the APVs for the first and the last site will be identical in the internal nodes, there is potential for data (APV entries) re-use.

Both entries only differ on $L_S^{(l)}$ and $L_S^{(r)}$, which by recursion depend only on the likelihood entries at the tip vectors. We can think of node p as a root node of a subtree (clade), whose likelihood is computed with data from a sub-alignment. If the data pattern at the tips of the subtree rooted at p are identical for sites i and j, then $L_A^{(p)}(i)$ and $L_A^{(p)}(j)$ must have the same value. Therefore, the likelihood entry only needs to be computed once. The same argument can be made for states `C, G` and `T`.

The key technical challenge with this approach is that it requires a large amount of bookkeeping, to keep track of identical subtree site patterns (for details see [96]). However, this approach can be simplified by considering only subtree-level presence of gaps in the alignment.

Gaps and undetermined characters are mathematically equivalent in the standard ML framework. Since structured patches of missing data dominate many phylogenomic datasets (the amount of missing data can be up to 90%) [74, 92], we only track subtree site patterns that entirely consist of gaps/undetermined characters (e.g., we are only interested in subtree sites of the from: `----` in a subtree of size 4). We assume that undetermined characters (`?`, `N`) have been translated into gap symbols (`-`). Thereby, we avoid the more complex task (see [64] and [100]) of tracking *all* identical subtree site patterns (e.g., detecting all sites of the form: `ACCT` in a subtree of size 4). We show an example in Figure 3.11

This restriction simplifies the required bookkeeping procedure and data structure significantly, because we only need to know whether a subtree site consists entirely of gaps or not. Thus, given an alignment with s sites, it suffices to enhance the data structures for storing tips and inner nodes by a

simple bit vector with s bits. If all-gap sites are represented by 1 and non-gap sites by 0, we simply need to execute a bit-wise `AND` on the respective bit vectors of the child nodes l and r in conjunction with the tree traversal for computing the likelihood to determine the all-gap sites at the ancestral node p (see Figure 3.12). We can then use this bit vector at p to determine if we need to compute the likelihood entries of the ancestral probability vector at a site i.

We have implemented this method for DNA and protein data under the Γ model of rate heterogeneity in RAxML v728 (alpha) available at `http://www.exelixis-lab.org/web/personal_page/izquierdo/sev.tar.gz`. Evidently, the efficiency of this approach depends on the proportion of and distribution of gaps in the input alignment. Since areas of missing data are typically well-structured in current phylogenomic datasets, this approach is expected to work well with this kind of input data.

While SEVs can speed-up ancestral probability vector computations, SEVs slightly slow down the branch length optimization and likelihood computation (at the root) functions because of the memory accesses to the bit vectors.

Saving Memory with SEVs

SEVs as implemented here, can also be deployed to reduce memory requirements. As mentioned above, if, at an ancestral node p we encounter an all-gap site, we completely omit its computation. In order to accomplish this, we need to maintain only one additional ancestral probability vector site, that contains the signal for all-gap sites. Consider an ancestral probability vector where 50% of the entries in the all-gap site bit-vector are set to 1, that is, where we only need to compute 50% of the ancestral probability vector entries with respect to the total alignment length.

We can observe that, in addition to saving 50% of the computations required for this ancestral probability vector, we can also save 50% of the memory space required for storing the ancestral probability vector (see Figure 3.13). Thus, the memory requirements for each ancestral node can be determined on-the-fly as we traverse the tree, by subtracting the number of entries that are set to 1 in the bit vector from the input alignment length. Remember that, the bit vectors we deploy are always as long as the input alignment.

The key technical problem that arises is that, the required ancestral probability vector lengths at inner nodes will change dynamically when the tree topology changes or even when the tree is just re-rooted. Given a rooting of the tree, one may think of this as ancestral probability vectors becoming longer while one approaches the root of the tree.

At present we have implemented this by dynamically freeing and allo-

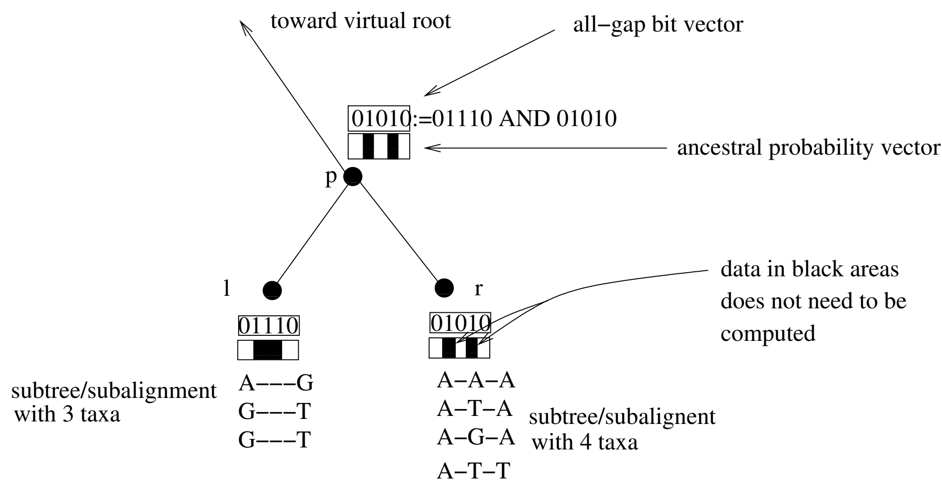

Figure 3.12: Using Subtree Equality Vectors to save computations for all-gap alignment sites in subtrees.

cating memory (using `free()` and `malloc()`) at each ancestral node. The reallocation only takes place when the all-gap bit-vector count (number of bits set to 1) corresponding to the required ancestral probability vector does not equal the all-gap bit-vector count of the current ancestral probability vector at an ancestral node. Due to the large number of calls to `malloc()` and `free()` performance can suffer when the pthreads version is being deployed due to thread contention. The technique may not scale well with the number of threads because the reallocation of shared memory by one thread can block other threads. One possible solution is the deployment of lockless memory allocators [34], where the syncronization overhead dissapears because each thread works with local allocation and memories.

Note that, the concepts presented here can also be applied to phylogenomic datasets with joint branch length estimates across partitions, while the conceptually different ideas presented in [92] can only be applied to partitioned phylogenomic datasets with per-partition branch length estimates.

3.4.2 Generation of Biological Test Datasets

To assess our methods, we used two large multi-gene datasets of plants.

The first dataset comprises 37,831 taxa and 9,028 sites and was obtained as follows: We assembled a DNA sequence matrix of 37,831 seed plant taxa consisting of the chloroplast regions atpB (1,861 taxa, >2.6 Megabases [Mb]), matK (10,886 taxa, >14.3 Mb), rbcL (7,319 taxa, >9.7 Mb), trnK (4,163 taxa, >7.5 Mb), and trnL-trnF (17,618 taxa, >13 Mb), and the internal

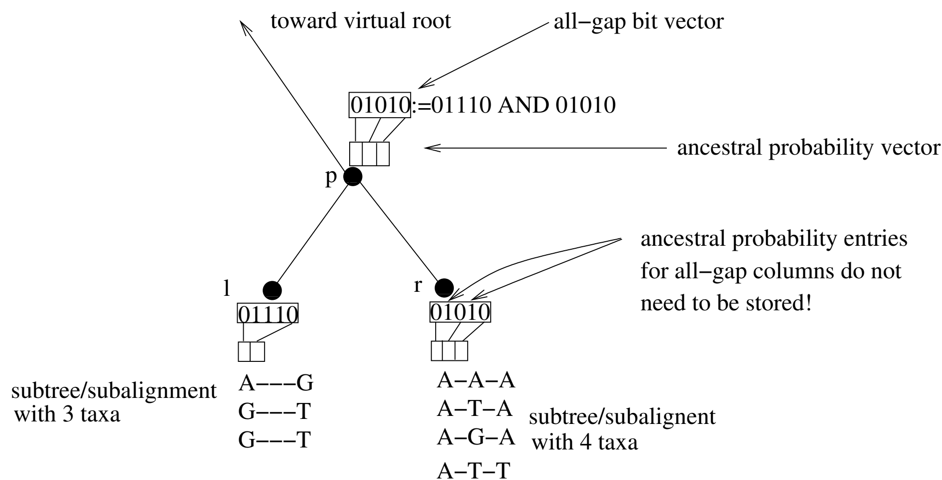

Figure 3.13: Using Subtree Equality Vectors to save computations *and* memory for all-gap alignment sites in subtrees.

transcribed spacer (ITS; 26,038 taxa, >14.3 Mb), using the Phylogeny Assembly with Databases tool (`PHLAWD` [81]). All sequence alignments were conducted using MAFFT version 6 [42] for initial alignments and MUSCLE for profile alignments [19]. Alignment matrix manipulations were performed with Phyutility [82].

The second dataset comprises 55,593 taxa and 9,853 sites and was obtained using the same pipeline as described above. The gene regions used were atpB (2,346 taxa, >3.6 Megabases [Mb]), matK (14,848 taxa, >33.6 Mb), rbcL (10,269 taxa, >14.9 Mb), trnK (5,859 taxa, >15.3 Mb), and trnL-trnF (25,346 taxa, >30.1 Mb), and the internal transcribed spacer (ITS; 37,492 taxa, >30.9 Mb).

For ease of reference we henceforth denote the 37,831 taxon datasets as 38K and the 55,593 taxon as 56K. Trees computed on the 56K dataset have been published [80].

3.4.3 SEV Performance

We used the 38K and 56K datasets to test memory savings and speedups achieved by applying the adapted SEV technique to phylogenomic datasets. The gappyness (percentage of missing data in the alignments) is 81.53% for 38K and 83.40% for 56K, respectively.

For each alignment, we computed a parsimony starting tree with RAxML that was then evaluated (model parameter and branch length optimization without tree search, RAxML `-f e` option) with RAxML under the GTR+Γ

	SEVs	SEVs with memory saving	standard
Runtime (s)	4125.1	4116.8	6541.1
Memory (GB)	42	15	41
LogLikelihood	-5528590	-5528590	-5528590

Table 3.3: Execution times and memory requirements for optimizing model parameters and branch lengths under on the 38K dataset using SEVs, SEVs with memory saving, and the standard likelihood implementation.

	SEVs	SEVs with memory saving	standard
Runtime (s)	7145.2	8095.1	11181.4
Memory (GB)	67	29	67
log likelihood	-7059556	-7059556	-7059556

Table 3.4: Execution times and memory requirements for optimizing model parameters and branch lengths under on the 56K dataset using SEVs, SEVs with memory saving, and the standard likelihood implementation.

model using the SEV reimplementation (with and without memory saving) and using the standard likelihood implementation.

The standard implementation required 41GB of memory on the 38K dataset and 66GB of memory on the 56K dataset. The SEV technique with the memory saving option enabled (`-U` option, available as of RAxML v7.2.7) reduced memory footprints under Γ to 14GB (38K) and 21GB (56K) respectively. The log likelihood scores for all three implementations were exactly identical. As shown in Table 3.3 and Table 3.4, the runtimes of the SEV-based versions are 25-40% faster than for the standard implementation. The runtime differences between the SEV-based implementation with memory saving enabled and the plain SEV version without memory saving, can be attributed to differences in memory access patterns. While both versions conduct the same number of computations, the memory-saving version needs to make millions of calls to OS routines (`free()` and `malloc()`) while the plain SEV version exhibits a higher memory footprint and thereby, potentially, a higher cache miss rate.

3.5 Summary

In this chapter we have described the memory requirements of phylogenetic inference under ML. Accommodating such huge memory requirements is necessary for analyzing phylogenomic datasets. We have presented several techniques to effectively reduce these requirements. This will allow for computing

the PLF on larger datasets than ever before, especially when the limiting factor is RAM memory.

We have presented the first implementation of the PLF that relies on out-of-core execution. We find that, given the locality of ancestral probability vector access patterns, miss rates are very low, even if the amount of available RAM is limited to a small fraction of the actually required memory. We demonstrate that our out-of-core implementation, performs substantially better than the standard implementation that relies on paging.

We have presented a generic strategy for the exact computation of log likelihood scores and ML tree searches with significantly reduced memory requirements. The additional computational cost incurred by the larger number of required ancestral vector recomputations is comparatively low when an appropriate vector replacement strategy is deployed. The memory versus additional computations trade-off can be adapted by the users via a command line switch to fit their computational resources. We also show that, the minimum number of ancestral probability vectors for computing the PLF that need to be kept in memory for a tree with n taxa is $log_2(n) + 2$. This result may be particularly interesting for designing equally fast, but highly memory-efficient phylogenetic post-analysis tools that rely on full tree traversals.

The MRC strategy has been integrated in RAxML-Light [91] and in the Phylogentic Likelihood Library [27]. This will allow to infer trees and compute likelihood scores on a single multi-core system datasets that previously would have required a supercomputer. The recomputation strategy clearly outperforms the out-of-core approach. For this reason, the later is not currently maintained in any production codebase.

We have adapted and re-implemented the SEV technique for phylogenomic datasets with missing data and enhanced it by a novel memory-saving option. This technique can reduce execution times by 25-40% on sufficiently 'gappy' datasets via omitting redundant computations. More importantly, the revised SEV technique can be deployed to achieve significant memory savings that are almost proportional to the amount of missing data in the test datasets. This technique has already been fully integrated into the standard RAxML distribution and is also available in ExaML [90].

On the software engineering side, it is important to note that the recomputation and the SEV memory-saving techniques are orthogonal. The recomputation technique reduces the number of APVs stored in memory. The SEV technique reduces the size of individual APVs. Therefore, both techniques can be applied simultaneously, and combined with potential future techniques such as the use of lossless compression algorithms for storing ancestral probability vectors.

Chapter 4

The Backbone Algorithm

> The content of this chapter has been derived from the following peer-reviewed publication:
> F. Izquierdo-Carrasco, S. Smith, and A. Stamatakis. Algorithms, data structures, and numerics for likelihood-based phylogenetic inference of huge trees. *BMC Bioinformatics*, 12(1):470, 2011
> S. Smith generated the 38K and 56K datasets described in Subsection 3.4.2

This chapter introduces an algorithm for space-constrained phylogenetic tree searches, which we denote as backbone algorithm. The notion behind this search algorithm is that, in large alignments, the sequences at the tips are close enough to each other to be grouped correctly in the early phases of the tree search algorithm. Therefore, it should be possible to devise some search heuristics that spend more time evaluating topological moves on inner parts of the tree, which are more likely to yield likelihood increases. Furthermore, by exploring a smaller tree space, the convergence criteria, as described in Section 2.6 should be met earlier. The backbone algorithm, which we describe in detail in this chapter, can reduce the time required for tree inferences by more than 50% while yielding equally 'good' trees in the statistical sense.

4.1 Constraining tree search to a backbone tree

PhyNav (Phylogenetic Navigator [104]) first introduced the idea to reduce the dimension of the tree for the search phase. PhyNav reduces the number

of sequences in the MSA to a subset that contains the most relevant phylogenetic information. The subtree based on the reduced alignment is faster to search on and optimize, and can be used as a scaffold to construct the full, comprehensive tree.

Similarly, we explored the idea of identifying closely related taxa and collapsing them to a subtree root, which can be considered as a single *virtual tip*, also called *super-taxon*. The tree induced by all virtual tips and remaining original (non-collapsed) tips is called *backbone* tree. This technique can also potentially reduce the memory footprint of the tree, because the number of internal nodes that are actually updated is reduced.

By clustering taxa into virtual tips, the dimension of the tree can be reduced allowing for a tree search on the backbone tree that is induced by the virtual tips. Given a perfectly balanced tree, a reduction of 50% corresponds to collapsing each pair of taxa into a single virtual tip. Thus, for each pair of tips, there is one less inner node to operate on, and the total number of inner nodes is halved. We henceforth denote such a reduction of the tree dimension as *reduction factor*.

Once an appropriate backbone tree has been computed, topological moves can be applied, following the standard tree search strategy such as the one described in Section 2.6. The difference with respect to the standard search is that the topological moves are restricted to the backbone tree.

In other words, the virtual tips are interpreted as tips in the backbone tree on which we conduct the tree search. In our RAxML proof-of-concept implementation [38], which deploys SPR moves, only subtrees that form part of the backbone tree are pruned and will exclusively be re-inserted into branches that lie within the backbone.

Despite restricting the tree search to the backbone, in our setup, we always compute the log likelihood score of the comprehensive tree during the backbone tree search. The log likelihood score of the comprehensive tree can be easily computed, because virtual tips are ancestral probability vectors that summarize the signal of the (excluded) real tips situated below the respective virtual tip.

As discussed in Section 3.1, the memory requirements for storing the ancestral probability vector representing a virtual tip are significantly higher than for storing a terminal taxon.

4.2 Algorithm

The main user parameter for the backbone tree algorithm is the desired tree size reduction factor R, where $0.0 < R < 1.0$. This parameter controls how

much the backbone tree shall be reduced in size. Ideally, the backbone tree will then comprise $n \cdot R - 2$ ancestral nodes and $n \cdot R$ backbone tips. Backbone tips may either be virtual tips (ancestral nodes) or real tips. Evidently, choosing very low values of R may significantly impact the quality of the inference, especially if very short branches are present. According to our experiments (see Section 4.3), using $R > 0.25$ is a conservative minimum value for R.

The algorithm comprises the following computational steps:

Starting tree A reasonable starting tree is given by the user or generated by a fast method, for instance, parsimony.

Tip Clustering We assign the n tips of the starting tree to $n \cdot R$ clusters, that is, $c = \lceil n \cdot R \rceil$, where c is the total number of clusters obtained. For each tip we store a cluster identifier that denotes to which cluster the tip has been assigned.

Backbone delimitation Determine and mark the ancestral probability vectors that will become virtual tips in the backbone. We traverse the tree and use the cluster identifiers to label all ancestral nodes as residing `inside`, `outside` or on the `boundary` of the backbone.

Backbone Tree Search Conduct a standard tree search (see Section 2.6) restricting topological moves to the backbone tree.

To also achieve a memory footprint reduction, one could write a multiple sequence alignment for the backbone to file that will partially consist of nucleotide sequences and partially of ancestral probability vectors representing virtual tips. This reduced alignment can then be parsed together with the backbone tree for conducting a tree search. This approach, however, has not been implemented.

We next outline the four algorithmic steps in detail.

4.2.1 Starting tree

To build a backbone, we assume that a reasonable (i.e., non-random) fully resolved comprehensive tree T comprising all taxa (e.g., obtained via parsimony using TNT [28], Parsimonator [85] or RAxML [87]) is computed or provided as input. This comprehensive n-taxon tree has n tips and $n-2$ ancestral (inner) nodes.

Once the parsimony tree is available, the branch lengths and the statistical model parameters are optimized under Maximum Likelihood.

4.2.2 Tip Clustering

We present an approach based on computing a distance matrix and applying average-linkage hierarchical clustering [55].

Hierarchical clustering with k small distance matrices

In standard hierarchical clustering, the first step consists of calculating a distance matrix that contains the pair-wise distances between all items (tips) to be clustered. However, given a comprehensive tree T with ML estimates of branch lengths, we can directly obtain this distance matrix from the tree by calculating the pair-wise *patristic distances*. The patristic distance between two taxa is the sum of branch lengths on the path in the tree connecting the two taxa. Thus, the distance matrix is symmetric. The space requirements for storing such a patristic distance matrix are in $O(n^2)$ which can become prohibitive for large alignments with $n \geq 30,000$ tips. We observe that, the pair-wise patristic distances between most tips will be very large and hence these tips will be assigned to different clusters anyway. Therefore, to save memory, one can decompose this process into computing several smaller, partial distance matrices, since the comprehensive starting tree already induces a hierarchical clustering structure. If we subdivide the problem into computing k partial pair-wise distance matrices, and each partial matrix i defines c_i clusters, we need to ensure that $c = \sum_{i=0}^{k} c_i = \lceil n \cdot R \rceil$, so that the total number of desired clusters still corresponds to the specified reduction factor R. To achieve this, we do not fix the number of partial matrices k a priori. Instead, we define a threshold value m that represents an upper bound for the number of tips contained in each partial matrix. Let n be the total number of taxa, n_i the number of tips in a partial matrix, where $n_i \leq m$ and $n = \sum_{i=0}^{k} n_i$. From each partial matrix, we extract an amount of clusters proportional to its size, that is, $c_i \propto c \times \frac{n_i}{n}$.

This is implemented as follows: First, we find a set of subtrees such that (i) each subtree has as many tips as possible and at most m tips and (ii) each tip is included in exactly one subtree, that is, all tree tips are included in one subtree and no tip forms part of more than one subtree.

For each such subtree i, we then build a (partial) patristic distance matrix for all n_i subtree tips. Thereafter, we cluster them, by generating a hierarchical cluster tree. This hierarchical tree may be cut at different levels to generate a varying number of subtree tip groups. We choose to cut the tree such that it generates c_i clusters of subtree tips. If required, the number of desired clusters c_i will have been iteratively adjusted beforehand (for further details see below) for each partial matrix i to ensure that $c = \sum_{i=0}^{k} c_i$.

For example, consider a $40,000$-taxon tree, a reduction factor of 0.5 (corresponding to $20,000$ clusters), and a partial matrix threshold of $32,000$ taxa. In this example, we may obtain distance matrices of $10,000$ and $30,000$ taxa respectively. Then we will need to extract $15,000$ clusters from the $30,000$ taxon distance matrix and $5,000$ clusters from the $10,000$ taxon distance matrix.

To be able to apply this method and compute partial patristic distance matrices, we need to devise an algorithm that selects subtrees from the comprehensive phylogeny such that they contain at most m taxa. We start by selecting the *innermost node* of the tree, as described below.

Computation of the innermost node

Each inner node i of an unrooted binary tree T is a trifurcation that defines three subtrees $T_{i,a}$, $T_{i,b}$ and $T_{i,c}$. We define the subtree length $stl(T_i)$ as the sum of all branch lengths in subtree T_i. Thus, $stl(T_{i,a}) + stl(T_{i,b}) + stl(T_{i,c}) = stl(T)$ holds for any inner node i, where T is the comprehensive tree.

In our current default implementation, we select the innermost node j that maximizes $stl(T) - \max\{stl(T_{j,a}), stl(T_{j,b}), stl(T_{j,c})\}$. An alternative criterion for selecting the innermost node is to determine the node that minimizes the variance of the three outgoing subtree lengths. Other possible criteria, that are not based on subtree length may be defined, for instance, as finding the node that minimizes the variance of the node-to-tip distance or finding the node with the highest minimum node-to-tip distance. The node-to-tip distance is defined as the sum of branch lengths on the path in the tree leading from an ancestral node to a tip.

The *tree diameter* is defined as the number of nodes on the longest path between any pair of tips. We define the *node distance* between two nodes as the number of nodes on the path that connects the nodes. The *normalized node distance* is defined as the raw node distance between alternative innermost nodes, divided by the tree diameter. We conducted an empirical assessment (based on our collection of large real-world trees) to compare the node distances between the innermost node generated by our criterion and the innermost node of these alternative approaches. Table 4.1 shows the results for a real-world dataset comprising 55,593 taxa (see Subsection 3.4.2). The respective innermost nodes (as identified by the alternative criteria) are either identical or close neighbors, that is, located in the same region of the tree.

Alternative criterion	Node distance	Normalized node distance
Lowest subtree length variance	0.00	0.00
Lowest node-to-tip distance variance	2.50	0.01
Maximal minimum node-to-tip distance	11.90	0.06

Table 4.1: Node-distances from the default criterion (55,593 taxa, averaged across 10 trees). Several criteria can be employed to select the innermost node of an unrooted tree. The alternative innermost nodes are located close to each other with respect to the tree diameter (186 nodes).

Tip clusters are delimited from k subtree roots

Once we have determined the innermost node, we conduct a depth-first tree traversal starting at this node and descending into each of the three subtrees. The depth-first traversal terminates, when a subtree root is encountered that comprises $\leq m$ tips. All subtree roots that contain $\leq m$ tips are stored in a list for further processing. Thus, when the depth-first traversal has been completed, this list of k subtree roots can be used to generate the k partial patristic distance matrices of maximum size $O(m^2)$. In our implementation, we set $m := 1024$. This is a suitable value, since the time required for processing partial distance matrices of such size remains in the order of seconds, which is negligible in comparison with total runtime of the tree search algorithm.

For each subtree root (i.e., each partial patristic distance matrix), we determine how many clusters should approximately be extracted, via $\bar{c_i} := \lfloor \frac{1}{2} + c \cdot \frac{n_i}{n} \rfloor$, where i is the subtree number, n_i is the number of tips in the respective subtree, $c = n \cdot R$ is the total number of desired clusters, and $\bar{c_i}$ is the number of clusters for subtree i. In general, $c \neq \sum_{i=0}^{k} \bar{c_i}$. The overhead (or deficit) of clusters, that is given by $\Delta c = c - \sum_{i=0}^{k} \bar{c_i}$, is then proportionally distributed across all remaining partial matrices. This process is repeated iteratively until no overhead (or deficit) remains. In each iteration, we reassign $c_i := \lceil \bar{c_i} + \Delta c \cdot \frac{\bar{c_i}}{c} \rceil$ until $c = \sum_{i=0}^{k} c_i$ for every i.

Then, for each subtree $i = 1...k$ we proceed as follows:

1. For all tips in subtree i, calculate the patristic distances to all other tips in this subtree and save them in the respective distance matrix.

2. Apply pairwise average clustering to generate a hierarchical tree of joins from the distance matrix.

3. Cut the tree, such that exactly c_i clusters are generated.

4. Add those clusters to a global list of clusters. Maintain a list that keeps track to which cluster a tip belongs.

4.2.3 Backbone construction

When all subtrees have been processed, we have a list of c clusters. Note that, each cluster contains x tips, where $1 \leq x \leq m$ and that each tip is assigned to exactly one cluster. The step to build the backbone from the clusters is not trivial. We use labels (`inside`, `boundary` and `outside`) to identify which nodes belong to the backbone and which ones do not.

The backbone tree is defined by nodes marked as `inside` and `boundary`. Once the clusters have been computed, we build the backbone as follows: Initially, we label each inner node in the tree as `inside`, tip nodes which belong to clusters of size one as `boundary`, and all remaining terminal nodes as `outside`. In addition, we maintain a list for storing the cluster identifiers of ancestral nodes that will not form part of the backbone.

Once this is done, we update/adapt the backbone assignment for ancestral nodes: The nodes of the comprehensive tree that represent the k subtree roots will remain inside the backbone. On each of the k subtree roots, we initiate a post-order traversal to relabel the ancestral nodes, if required, according to the following rule set:

- If the two child nodes are labeled as `inside` or `boundary`, the ancestral node remains labeled as `inside`.

- If one child is labeled as `inside` or `boundary` and the other child as `outside`, the ancestral node is relabeled as `inside` and the `outside` child node is relabeled as `boundary`.

- If both children are labeled as `outside`, we need to check to which cluster they belong. If they belong to the same cluster, the parent node is labeled as `outside` and the shared cluster identifier of the child nodes is propagated to the parent node. If the two children do not belong to the same cluster, the parent node is labeled as `inside` and both children are relabeled as `boundary`.

When the post-order traversal is about to be completed, we arrive at the subtree root i again, which was originally labelled as `inside`. At this point, we check whether the adjacent backbone node of the subtree root i has been labeled as `outside`. Whenever this is the case (see Figure 4.1 for an example), the adjacent backbone node is relabeled as `boundary` for consistency.

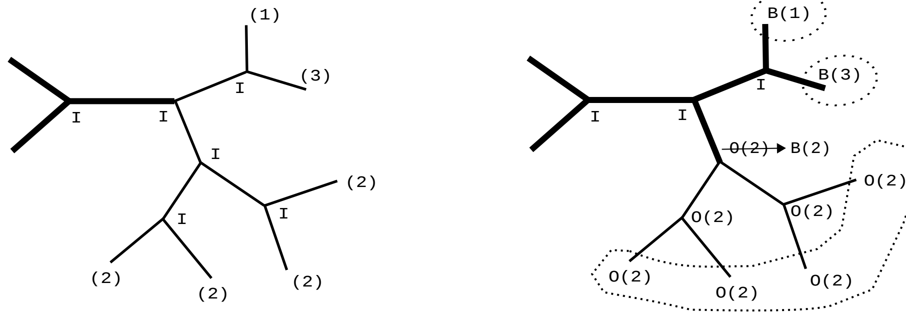

Figure 4.1: Consistency of labels at the backbone boundaries. At first (left) an initial backbone exists (thick branches), all inner nodes are labelled as inside (I) and each tip node has a cluster id. After completion of the post-order traversal (right), each inner node has been relabelled accordingly, if required. Here, cluster 2 is monophyletic, hence the cluster id was inherited propagated back to the initial backbone node. This produced a branch (edge) with an inside and an outside node; therefore the outside(O) node is relabelled (arrow) as boundary(B) node.

Reduction Factor R	$n := 37831$ (expected)	$n := 55593$ (expected)
0.25	12668.0 (9457.75)	19366.7 (13898.25)
0.50	22340.0 (18915.5)	33501.5 (27796.5)

Table 4.2: Average number of computed backbone tips over 10 distinct trees. The average number of backbone tips is higher than the expected number $n \cdot R$

Given a set of tips that form part of the same cluster, it may occur that these tips also form a monophyletic group. In this case, during the post-order traversal, all ancestral nodes will be grouped together under the same cluster identifier and the common ancestral node will become a backbone `boundary` (virtual tip). However, if the tips in a cluster are not monophyletic (see for instance, in Figure 4.2), the application of the above rules requires some additional `boundary` relabelling.

Based on the prolegomena, a single cluster may thus induce more than a single virtual tip. As a consequence, the number of virtual tips may actually be higher than the number of clusters. In turn, the reduction of tree size that can be achieved will be smaller than specified by R. The impact and frequency of occurrence of this phenomenon (non-monophyletic clusters) depends on the shape of the tree and the branch lengths. In Table 4.2, we outline this effect for trees with 37,831 and 55,593 taxa. We computed the average number of virtual tips generated by our algorithm on 10 distinct trees per dataset and reduction factors of 0.25 and 0.5 respectively.

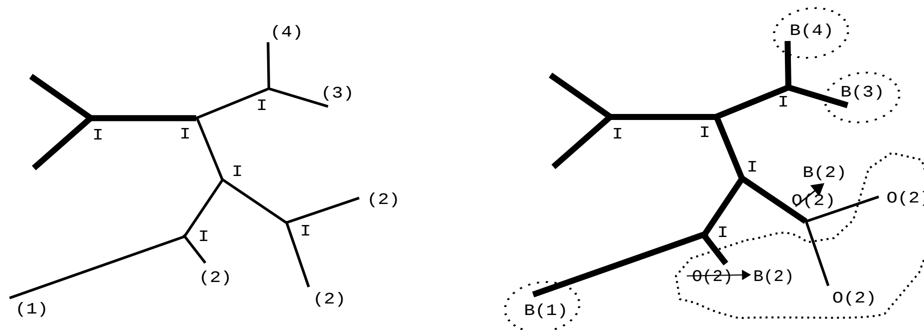

Figure 4.2: At first (left) an initial backbone exists (thick branches), all inner nodes are labelled as inside (I) and each tip node has a cluster id. Upon completion of the post-order traversal (right), each inner node has been relabelled accordingly. Here, cluster 2 is not monophyletic. Hence, an additional virtual tip is created, that is, cluster 2 generates 2 boundary tips.

4.2.4 Backbone-constrained Tree Search

Once the Backbone has been built, a standard phylogenetic search algorithm can be applied. We have implemented the above backbone algorithm in a dedicated proof-of-concept RAxML version. This implementation is available for download at `http://www.exelixis-lab.org/web/personal_page/izquierdo/backbone.tar.gz`

Initially, RAxML will generate a comprehensive randomized stepwise addition order parsimony tree, or read in a user specified tree via `-t`. Then it will optimize ML model parameters—including branch lengths—on the comprehensive tree. Thereafter, it will execute the backbone algorithm as described above. The tree searches on the backbone are based on the standard RAxML hill-climbing algorithm. The standard algorithm implements *lazy SPRs* steps, because the likelihood scores obtained are only approximate. Like in standard SPRs (see Section 2.6), a subtree is pruned and re-grafted on several neighbouring branches. However, after re-grafting, only the three branch lengths around the insertion branch are re-optimized, which induces significant time savings in comparison with re-optimizing all branch lengths of the tree. These lazy SPRs are used as a fast pre-scoring mechanism to find good topologies which are later optimized more thoroughly [95] in a subsequent evaluation step. In our algorithm, lazy SPR moves are only conducted within the backbone.

After each complete cycle of SPR moves (see [87] for details), the backbone tree will be re-computed based on the currently best tree. Also, the branch lengths of the entire tree (including those branches not forming part

of the backbone) will be re-optimized once after each SPR cycle.

4.3 Evaluation and Results

4.3.1 Performance

To test the backbone algorithm we executed the dedicated RAxML version. with the experimental `-L` command line option. This option initially builds a backbone tree and then deploys the CAT approximation of rate heterogeneity [86] with the standard RAxML hill-climbing search algorithm [87, 93] to apply lazy SPR moves (see [87]) within the backbone only. We used tree size reduction factors of 0.25 and 0.5. As starting trees, we used randomized stepwise addition order parsimony starting trees generated with RAxML v727 (`-y` option). For each dataset, we inferred 10 ML trees for each of the 10 parsimony starting trees. RAxML was executed using the Pthreads-based parallel version [97] with 16 threads on unloaded Quad-Core AMD Opteron nodes with 16 cores and 128GB RAM each.

We computed average runtimes over 10 runs for the 38K and 56K datasets (see Subsection 3.4.2) respectively. For each backbone tree, we also computed the theoretical minimum number of bytes (denoted as *Memory for Backbone*) required to store the ancestral probability vectors at the virtual tips and the inner nodes which dominate memory requirements. If the branch length optimization process, unlike in our current implementation, is limited to optimizing branches within the backbone, this theoretical minimum value represents a good estimate of the memory footprint for a backbone tree search. We also computed the respective memory requirements for the comprehensive tree (denoted as *Memory for Full tree*), which reflect the 'standard' memory requirements when no reduction factor is applied.

These values (see Table 4.3 and Table 4.4) provide a notion of the potential memory savings that can be achieved by the backbone approach. In Table 4.3 and Table 4.4 we also provide the respective execution times and average log likelihood scores obtained by using the backbone algorithm ($R := 0.25$, $R := 0.5$) and a comprehensive search on the full tree ($R := 1.0$). Those values have been averaged over 10 runs (10 starting trees). While execution times can be reduced by the backbone approach, log likelihood scores obtained by conducting searches on a backbone are slightly worse than those obtained by searching on the full tree.

In Figure 4.3 and Figure 4.4 we show that the choice of the random number seed (`-p` option in RAxML), that determines the shape of the starting trees, has a significant impact on the final log likelihood score (computed

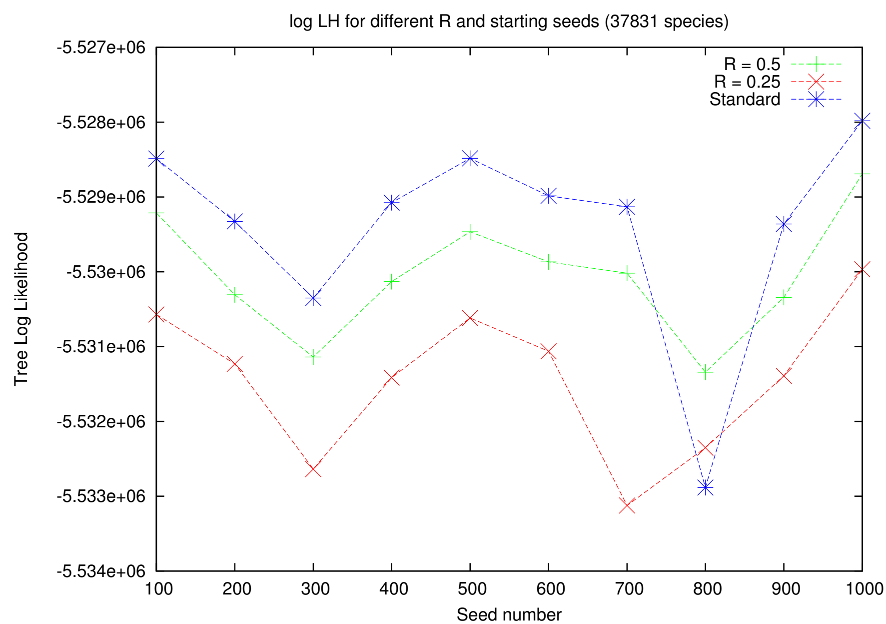

Figure 4.3: Log Likelihood scores for different Reduction factors (38k dataset). Plot of log likelihood scores under GTR+Γ of the final trees obtained by each method as a function of the starting tree (random number seed) for the 38K dataset. Each LH score (point) results from an independent search. The lines linking the points are only guiding the eye.

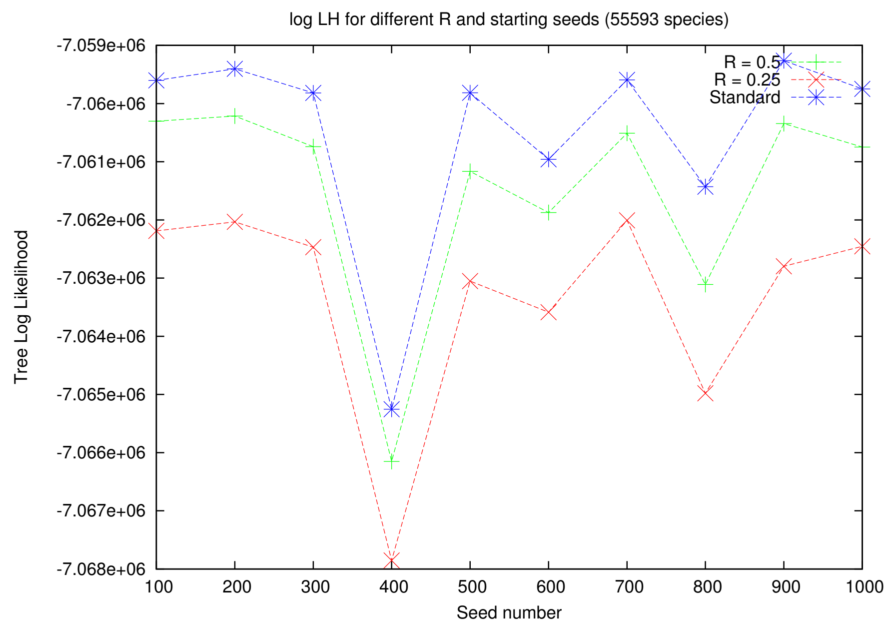

Figure 4.4: Log Likelihood scores for different Reduction factors (56k dataset). Plot of log likelihood scores under GTR+Γ of the final trees obtained by each method as a function of the starting tree (random number seed) for the 56K dataset. Each LH score (point) results from an independent search. The lines linking the points are only guiding the eye.

	R=0.25	R=0.5	R=1
Runtime (h)	30.41	38.60	54.03
Memory for Backbone (GB)	4.90	7.70	N/A
Memory for Full tree (GB)	10.33	10.33	10.33
LogLikelihood (Avg)	-5531436	-5530051	-5529406
LogLikelihood (Std Dev)	943.26	770.47	1307.16
Avg (logLH - logLH(R=1))	2030.24	645.31	0.0

Table 4.3: Average runtimes, memory requirements, and log likelihood scores (over 10 runs) for the 38K dataset.

	R=0.25	R=0.5	R=1
Runtime (h)	50.17	63.22	85.89
Memory for Backbone (GB)	8.22	12.72	N/A
Memory for Full tree (GB)	16.82	16.82	16.82
LogLikelihood	-7063342	-7061516	-7060488
LogLikelihood (Std Dev)	1727.90	1761.27	1718.47
Avg (logLH - logLH(R=1))	2853.41	1028.04	0.0

Table 4.4: Average runtimes, memory requirements, and log likelihood scores (over 10 runs) for the 56K dataset.

under GTR+Γ), irrespective of the search strategy that is used. On average, searches on the full tree yield better likelihood scores than searches on backbone trees. However, the variance of the likelihood score as a function of the starting tree (parsimony random number seed) is analogous to the score variance between full and backbone tree searches. For example, on the 38K dataset, the log likelihood scores on 10 final trees obtained for full searches show a standard deviation of 1307 log likelihood units. The average difference in log likelihood scores per starting tree between the full search and a backbone search with $R := 0.50$ is only 645 log likelihood units and 2030 log likelihood units for backbone searches with $R := 0.25$, respectively.

Given the runtime savings that can be achieved by the backbone approach, backbone tree searches can be used, for instance, to explore a larger number of parsimony starting trees which substantially influence the final log likelihood scores. A reasonable strategy for finding best-known ML trees may consist in starting many fast searches with a relatively aggressive setting of $R := 0.25$ to identify/determine a set of 'good' starting trees that yield the best final log likelihood scores. In a second step, full tree searches can be conducted on those promising starting trees to find trees with even better scores.

	$R := 0.25$	$R := 0.5$	$R := 1$
$R := 0.25$	182.6	169.9	188.0
$R := 0.5$	169.9	152.8	146.2
$R := 1$	188.0	146.2	133.0
True Tree	398.8	382.0	388.0

Table 4.5: Average symmetric differences (over 5 runs) for the 1500 simulated dataset.

4.3.2 Simulated Datasets (Accuracy)

We used simulated datasets in order to better understand the impact of the backbone algorithm on topological accuracy. We ran INDELible [26] to generate simulated MSAs of 1500 taxa (575 bp) and 5000 taxa (1074 bp). We compared the RF distance [69] (number of bipartitions that differ between two topologies) between the true tree and the topologies from the standard full search and the backbone-based ones. For each dataset, the full search and the backbone search with $R := 0.25$ and $R := 0.5$ were ran five times with different starting trees. Table 4.5 shows the average symmetric differences among all approaches for the dataset with 1500 taxa. We see that, in terms of topological accuracy, applying reductions of $R := 0.25$ and $R := 0.5$ yield topologies that are close to the standard full search. Furthermore, the distance to the true tree is not increased by the reduction.

4.4 Summary

In this chapter we have proposed an algorithm for reducing the tree size for phylogenetic inference under likelihood-based methods on trees with several tens of thousands of taxa. We have explored different backbone construction techniques and described the method that worked best with respect to final log likelihood scores. Such backbone-based techniques can help to reduce memory footprints and execution times. However, in almost all cases they yield final trees with worse likelihoods compared to comprehensive tree searches on a full, unreduced tree. We find that likelihood scores of final trees heavily depend on the respective starting trees and conclude that backbone approaches can be deployed for identifying 'good' starting trees, that can then be further refined using a comprehensive tree search.

Chapter 5

Introduction to GPU Programming

In this chapter, we briefly introduce the main programming models for GPU computing, where parallel compute-intensive calculations are offloaded to the GPU, while the rest of the code is run on the CPU.

5.1 Overview

GPUs (Graphics Processing Units) were originally designed as a dedicated hardware architecture for accelerating graphics rendering. GPGPU (General-Purpose computation on GPUs) [62], or *GPU computing*, refers to the use of GPUs to accelerate scientific applications, a concept that dates back to the early 2000s [102].

Modern CPUs consist of a few cores optimized for serial processing, while GPUs contain thousands of small efficient cores designed for parallel floating-point operations because of the computational requirements of graphics rendering.

The main programming models/interfaces for GPUs are CUDA (Compute Unified Device Architecture) and OpenCL. CUDA is a parallel computing (GPUs) platform introduced by NVIDIA in 2006. It includes a SDK and API that, using the C language, gives access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.

OpenCL (Open Computing Language) is as a more generic framework. It was initially introduced in 2009 and is currently maintained by the Khronos group [14]. OpenCL programs can be executed across heterogeneous platforms, such as central processing units (CPUs), graphics processing units (GPUs), and digital signal processors (DSPs). OpenCL includes a language

(based on C99) for writing kernels (functions that execute on OpenCL devices), plus application programming interfaces (APIs) that are used to define and then control the platforms.

The high level APIs of CUDA and OpenCL allow the application programmer to create C programs that can execute specified functions (Kernels). These Kernels can be run on the GPU's streaming processors.

In both platforms, the master system that steers the computations is denoted as *host*, which is usually a CPU. In addition, one or several *devices* are available. A device is a massively parallel processor with a large number of arithmetic and floating-point processing units. In terms of memory, both approaches assume a similar memory hierarchy, although the terminology differs.

5.2 CUDA

In this Section, we introduce the main terminology and concepts of the CUDA framework [61].

5.2.1 CUDA Hardware and Architecture

From a hardware architecture perspective, NVIDIA GPUs consist of scalable arrays of multi-threaded *Streaming Multiprocessors* (SMs). A group of threads running on the same processor core is called a *thread block*. The number of threads in a thread block is limited by the hardware. On current GPUs, a thread block can contain up to 1024 threads [61]. Furthermore, thread blocks can be scheduled in any order because they are required to execute independently. During a CUDA program execution, as thread blocks complete their execution, new blocks are launched in the vacant streaming multiprocessors. Finally, blocks are organized in one, two or three-dimensional *grids of threads*.

We now briefly introduce the NVIDIA Fermi architecture [11]. The Fermi architecture (see Figure 5.1) includes up to 512 CUDA cores, which are distributed across 16 streaming multiprocessors, each with 32 CUDA cores.

Each streaming multiprocessor (see Figure 5.2) has 16 load/store units, four SFUs (special function units), and 64KB of on-chip memory which is used as shared memory and L1 cache. A streaming multiprocessor manages and executes threads in groups of 32 parallel threads (warp). A warp executes one common instruction at a time.

A coherent L2 cache of 768KB is shared across all multiprocesors in the GPU. The host interface connects the GPU to the CPU via a PCI Express

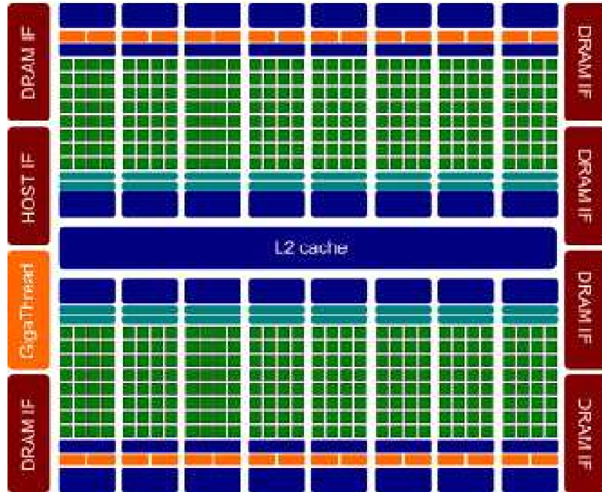

Figure 5.1: The Fermi Architecture (left) contains 16 streaming multiprocessors. Source [11].

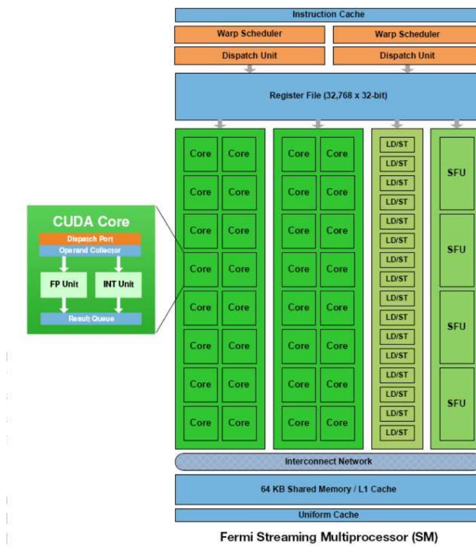

Figure 5.2: Architecture of a Fermi streaming multiprocessor. Source [59].

bus.

The SIMT (Single Instruction, Multiple Threads) and SIMD (Single Instruction, Multiple Data) architectures are closely related, since a single instruction is executed on multiple processing elements on different data or a data vector. In SIMT architectures (GPUs), the programmer can write thread-level parallel code for independent threads, as well as data-parallel code for coordinated threads. However, with respect to performance, it is important to avoid divergence among threads belonging to the same wrap, because a thread can either execute the same instruction as the other threads in the wrap, or idle. Therefore, for maximum efficiency, all threads of a wrap should share the same execution path.

In 2012 NVIDIA released a newer architecture called Kepler [60].In the course of this thesis, however, we only made use of the Fermi architecture.

5.2.2 CUDA Programming Model

In the CUDA programming model, the *host* is a CPU running the main (serial) C program. The host has its own RAM memory. The *CUDA threads* are executed on a separate *device* (GPU).

A *Kernel* is a CUDA C (an extension of C) function, which is executed in parallel N times by N different CUDA threads.

Barriers are used to synchronize thread blocks and coordinate memory accesses to global memory. Figure 5.3 shows this memory hierarchy in detail: Each thread has a private *local memory.* Each thread block has *shared memory*, that is visible only to threads within the same block (memory near the corresponding processor cores). All threads (and thread blocks) have access to *global memory.*

The CUDA programming model assumes that the device has its own DRAM memory (device memory). Therefore, the programmer is responsible for managing the memory units that are visible to kernels through calls to the CUDA runtime. This includes memory allocation and deallocation on the device, as well as data transfer between host and device memory.

5.2.3 Performance Considerations

Latencies should be hidden The latency is the number of clock cycles that a warp is waiting before executing its next instruction. This can happen when the input operands of the next instruction are not available yet. Another reason can be that the wrap is waiting due to a memory fence or a synchronization point. Each GPU multiprocessor can in principle hide these latencies and maximize the utilization of its functional units. If all wrap

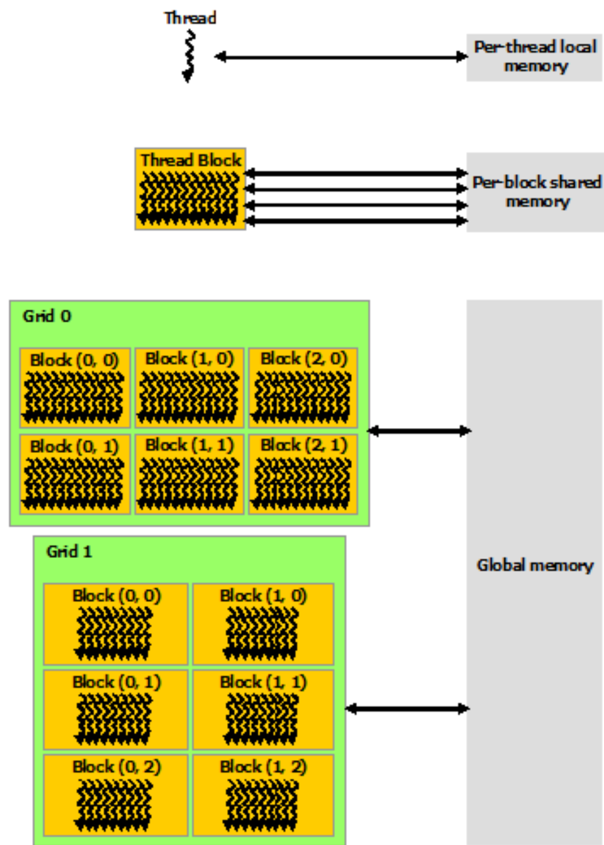

Figure 5.3: CUDA memory hierarchy. Source [61]

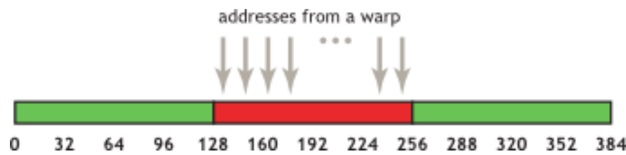

Figure 5.4: This access pattern results in a single 128-byte transaction, indicated by the red rectangle. Source [61]

schedulers always have at least one instruction to issue for one wrap at every clock cycle, the latency can be fully hidden. In other words, ideally a wrap will be issued at every cycle, and it is therefore desirable to have a high number of resident wraps, so that these can be alternatively scheduled.

Data transfer between Host and Device The amount of data transfer between host and device should be whenever possible minimized due to the low bandwidth. A typical approach is to sacrifice parallelism in the kernels by moving more code from the host to the device. This involves adding kernel code and data structures to compute and store intermediate results on the GPU DRAM that never need to communicate with the host. Transfer overheads can be minimized by batching many small transfers into single larger transfers.

Accesses to global memory Global memory resides in device memory, which is accessed via 32-, 64-, or 128-byte memory transactions. Segments of device memory (32-, 64- or 128-bytes) are aligned if their first address is a multiple of their size. If threads within a warp access such neighboring aligned memory elements, only one memory transaction is needed. In other words, data memory layouts should be designed by the programmer so that the warps read or write contiguous memory elements, as depicted in Figure 5.4.

5.3 OpenCL

The programming model of OpenCL is analogous to the one presented for CUDA. We briefly present it to clarify differences in terminology.

OpenCL (Open Computing Language) is an open standard for parallel programming of heterogeneous systems. It provides a language (a subset of ISO C99) for software developers to write portable code on SIMT (Single Instruction, Multiple Threads) architectures.

The OpenCL Execution Model consists of an application running on a Host (CPU), which offloads work to one or more Compute Devices (for instance GPUs). Each compute device is composed of one or more Compute Units. In CUDA, these are called Streaming Multiprocessors (SM).

A Kernel represents the code for a work-item (thread). Work-items are the basic units of work. Work-items are grouped into local work-groups equivalent to CUDA thread blocks. OpenCL applications can access various types of memory: Host memory (on the host CPU), `global` (visible to all work-groups, e.g., DRAM on the GPU board), `local` (shared within a work-group, called `shared` in CUDA), and `private` (registers per work-item, called local in CUDA).

5.3.1 OpenCL performance portability

OpenCL provides developers portability by enabling the usage of and deployment on diverse processing platforms. In particular, performance differences with CUDA are rather small on GPUs (CUDA performs at most 30% better), and tend to disappear under fair comparisons [22].

The OpenCL standard also guarantees functional compliance with other devices such as CPUs, DSPs, and other hardware platforms, that is, OpenCL portable code will run correctly and generate the same results. Thus, the main advantage is that a single implementation can be executed on different platforms. However, performance portability is not guaranteed. In order to obtain maximum performance, OpenCL code still requires architecture-specific implementations.

Recently, Zhang, Sinclair and Chien [114] studied the performance portability of OpenCL across diverse architectures (NVIDIA GPU, Intel Ivy Bridge CPU, and AMD Fusion APU) with typical benchmarks, such as SpMV (Sparse Matrix Vector multiply) and FFT (Fast Fourier Transform). The results showed poor performance for single-source OpenCL programs (on average 15% performance in comparison with state-of-the art implementations). However, with architecture-oriented the performance was improved to 67% of the performance on the Ivy Bridge CPU [114]. In general, there is a significant gap between single-source OpenCL programs and architecture-oriented tuned OpenCL programs.

Chapter 6

GPU implementation of Phylogenetic Kernels

The content of this chapter has been derived from the following peer-reviewed publication:
F. Izquierdo-Carrasco, N. Alachiotis, S. Berger, T. Flouri, S. P. Pissis, and A. Stamatakis. A generic vectorization scheme and a gpu kernel for the phylogenetic likelihood library. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2013 IEEE International Symposium on*, 2013
Simon Berger designed and implemented the generic Vectorization scheme described in Section 6.2.

In this chapter, we describe in detail a GPU implementation for the computation of the main phylogenetic functions as a proof-of-concept implementation for the Phylogenetic Likelihood Library (PLL, introduced in Section 2.7). These functions involve the computation of the likelihood score and the Newton-Raphson method for branch length optimization. The proof-of-concept implementation works for DNA data and the Γ model of rate heterogeneity.

We also introduce a GPU-specific memory organization scheme that reduces data transfer between the GPU and the CPU to an absolute minimum, thereby improving performance. The memory layout of the ancestral probability vectors (APVs) stored in the GPU is an adapted version of a *generic vectorization scheme*.

This generic vectorization scheme for the phylogenetic function (PLF) allows to transparently deploy vector units of arbitrary length for PLF computations. These vector instruction can be x86 intrinsics (128-bit wide SSE3

instructions and 256-bit wide AVX instructions) as well as SIMT instructions on GPUs. A generic vectorization scheme is important to ensure portability of the code to increasing vector lengths (e.g., the 512-bit wide vector units on the recent Xeon Phi processor).

According to our experiments, our GPU implementation of the PLF (Phylogenetic Function) is approximately twice as fast as the highly tuned x86 version of the PLF that relies on manually inserted AVX vector intrinsics.

The remainder of this chapter is organized as follows: In Section 6.1 we survey related work on PLF libraries and GPU implementations. Thereafter, in Section 6.2, a generic vectorization scheme for PLF computations is introduced. In the subsequent Section 6.3 we cover technical details of the GPU implementation. Thereafter, we describe the experimental setup and the results obtained (Section 6.4) and conclude in Section 6.5.

6.1 Related work

Early work on porting the RAxML likelihood functions, which comprise the core of the PLL, to GPUs in the pre-CUDA and pre-OpenCL era was reported in [12]. Exploiting fine-grain parallelism with GPUs for the PLF has previously been addressed in [65] and [115] for MrBayes [73]. However, these implementations represent case studies or only cover a small subset (for specific data types such as DNA data) of the PLF in MrBayes. Hence, these initial efforts do not represent production-level implementations, but rather proof-of-concept studies.

The BEAGLE [4] (general purpose library for evaluating the likelihood of sequence evolution on trees) library introduced an application programming interface (API) for PLF computations and also offers efficient implementations thereof. BEAGLE can exploit modern hardware using SSE3 intrinsics, multi-threading, and GPUs. It has been integrated into Bayesian programs (BEAST [99] and MrBayes [73]) and Maximum Likelihood programs (GARLI [116]). The BEAGLE paper [4] reports performance results for DNA and Codon data on two 15-taxon datasets. The test datasets contained 8558 unique nucleotide (DNA) site patterns and 6080 unique codon site patterns, respectively. For each of the three programs integrated with BEAGLE, the authors measured the speedup of the BEAGLE CPU, SSE3, and GPU (under single and double precision) implementations with respect to the corresponding native implementations. The largest speedups were obtained for the GPU implementation. For GARLI, only GPU speedups were reported (factor 3.8 for DNA data and 12 for codon data under double precision). The BEAGLE-based version of MrBayes yielded a maximum speedup

of 16 (DNA data) and of 31 (Codon data) on the GPU using double precision arithmetics. Note that the relative speedup for MrBayes comparing the BEAGLE CUDA against the BEAGLE SSE3 performance was approximately 4.6 for DNA data. BEAST showed similar speedups for the GPU implementation under double precision (14-fold for DNA data and 37-fold for codon data). The speedups for single precision were larger. However, for large-scale real-world datasets (in particular with a high number of taxa), double precision arithmetics are typically required to guarantee numerical stability of the PLF [7].

The PPL library introduced in Section 2.7, which we use to develop our GPU proof-of-concept implementation, offers additional features that BEAGLE does not support. The PLL can also use AVX intrinsics. Furthermore, it implements numerical optimization functions such as for instance the Newton-Raphson method for branch length optimization. BEAGLE defers these tedious programming tasks to the application programmer. It only offers functions for computing the first and second derivative of the likelihood function that can then be used by the application programmer to implement a Newton-Raphson branch length optimization procedure. Moreover, BEAGLE does currently not allow for conducting partitioned analyses which, given that partitioned analyses (distinct sets of likelihood model parameters are estimated for different parts of the multiple sequence alignment) are becoming increasingly common, represents a drawback of BEAGLE. As a consequence, BEAGLE does also not implement techniques [98, 113] for improving parallel load balance for partitioned analyses. Unlike the PLL, it does not offer a fine-grain MPI parallelization of the PLF and is hence limited to stand-alone shared memory nodes. Finally, BEAGLE does not implement the PSR (originally CAT) model of rate heterogeneity [86], which can yield substantial computational savings in terms of floating point operations *and* memory utilisation compared to the standard Γ model of rate heterogeneity [110].

6.2 Generic Vectorization

An important part of the PLF is the `newview()` function (see Section 2.7), which updates the ancestral probability vectors (APVs) at inner nodes of the tree in the course of a post-order traversal. The innermost loop of `newview()` calculates the sum over products between elements of the transition proba-

Node v			
Site 1		Site 2	
Rate 0	Rate 1	Rate 0	Rate 1
$L_A L_C L_G L_T$	$L_A L_C L_G L_T$	$L_A L_C L_G L_T$	$L_A L_C L_G L_T$

Figure 6.1: Memory layout of an ancestral probability vector

Node v	
Site 1/2 interleaved	
Rate 0	Rate 1
$L_{1,A}L_{2,A}L_{1,C}L_{2,C}L_{1,G}L_{2,G}L_{1,T}L_{2,T}$	$L_{1,A}L_{2,A}L_{1,C}L_{2,C}L_{1,G}L_{2,G}L_{1,T}L_{2,T}$

Figure 6.2: Memory layout of an ancestral probability vector with a vector width of 2 ($VW := 2$)

bility matrix P and corresponding elements in the APV L.

$$L_A^{(p)}(i) = \big(\sum_{S=A}^{T} P_{AS}(b_{lp})L_S^{(l)}(i)\big)\big(\sum_{S=A}^{T} P_{AS}(b_{rp})L_S^{(r)}(i)\big) \tag{2.10}$$

We assume DNA data and the Γ model of rate heterogeneity. The entries of these vectors are computed according to Equation 2.10 and stored following the layout shown in Figure 6.1, assuming DNA data with 4 states, 2 Γ rates, and 4 alignment sites.

For each alignment site, the ancestral probability vector contains 2 rate blocks. Each rate block contains 4 probabilities (one per state and site s, denoted by $L_{s,A}$, $L_{s,C}$,$L_{s,G}$, and $L_{s,T}$, for site s). Using this memory layout, the probability values of the states can be read efficiently from contiguous memory locations to calculate the scalar products in Equation 2.10.

This memory layout is used directly in the initial *ad hoc* SSE3 and AVX versions of RAxML. The calculation of the scalar products in the innermost `newview()` loop can be implemented by using element-wise multiply and horizontal add operations. However, this approach is only efficient, if the number of states (e.g., 4) is equal to or larger than the width of the vector unit. Since we use double precision floating point numbers this is the case both for SSE3 (vector width: 2), as well as AVX (vector width: 4) vector units. In contrast to this, modern GPUs have much wider vector units. In addition, the width of x86 vector units is also expected to increase (e.g., Intel Xeon Phi). Hence, the initial *ad hoc* vectorization scheme can no longer be used for the PLL. Moreover, the manual vectorization for each model and data type combination is error-prone and labor-intensive. Thus, we devised

a more generic vectorization scheme that is easier to port to new models and can conveniently be adapted to vector units of arbitrary length.

In order to use the wider vector units on GPUs, we introduce a new and more generic, vectorization scheme. Instead of exploiting parallelism within the innermost loop iteration of `newview()`, the new scheme now calculates a part of the ancestral probability vectors simultaneously for multiple sites. We denote this approach as *across-site vectorization*. In principle, across-site vectorization is analogous to the sequential implementation of the PLF: The calculations of the scalar products in the innermost `newview()` loop are carried out sequentially. The main difference is that the scalar products are now being calculated for multiple sites (i.e., 2 or 4 sites for SSE3/AVX or more than 64 sites on the GPU) in parallel. In the SSE3 and AVX implementations, this parallelism is exploited by using vector intrinsics. A similar scheme has been previously used for the inter-sequence vectorization of the PaPaRa 2.0 [8] dynamic programming algorithm.

This simple vectorization scheme can only be used when the data (i.e., the APVs) are stored using an appropriate memory layout. Such a layout needs to allow for reading the probability values of a specific state and rate (e.g., $L_{s,A}$) that belong to neighboring alignment sites from contiguous memory locations, that is, we require that $L_{s,A}$ and $L_{s+1,A}$ (for a given rate) occupy contiguous positions in memory. Since this is not possible using the standard memory layout (see Figure 6.1), we introduce an appropriately adapted and flexible (regarding the vector unit width) memory layout (see Figure 6.2) .

To assess the efficiency of this more generic vectorization scheme, we implemented it on both CPUs (using SSE as well as AVX) *and* GPUs. The major change consists in an adapted memory layout for the APVs which now allows to efficiently exploit across-site parallelism on CPUs and GPUs. Note that, SSE and AVX instructions currently do not offer efficient operations for loading data from non-contiguous sites (data locations) into vector registers (see Figure 6.1). Generally, GPUs offer greater flexibility with respect to loading values from non-contiguous memory locations (e.g., loading the values corresponding to state `A` and discrete Γ rate 0 of sites $0, 1, \ldots, 32$). However, overall GPU performance can be increased by accessing values from contiguous global memory locations, because read/write accesses can be coalesced and delays related to bank conflicts can be avoided. We have therefore generalized the ancestral probability vector memory layout to store corresponding values from different sites in contiguous memory. This allows for accessing the data at contiguous memory locations for the vectorized version of Equation 2.10. The memory layout is parameterized by the desired vector unit width (VW). For $VW := 1$, the memory layout corresponds exactly to the original memory layout of RAxML (see Figure 6.1). The analogous

layout for $VW := 2$ is shown in Figure 6.2. As outlined in Figure 6.2, corresponding values from different alignment sites (e.g., state `A`, discrete Γ rate 0 of sites 0 and 1) are located at contiguous memory locations. Therefore, they can directly be loaded into an SSE register via a single load operation. Given this altered and adaptive memory layout, implementing a vectorized version of Equation 2.10 for sites 0 and 1 becomes straight-forward. In Section 6.3 we show how we adapt this scheme to arbitrary widths for GPUs.

However, there do exist some limitations. Equation 2.10 can only be vectorized for sites that evolve according to the same model of evolution. In other words, the sites need to share the same P matrices and the same α shape parameter that determines the form of the Γ curve. With partitioned datasets, the parameters of the model of evolution will be optimized independently for each partition. Thus, the maximum vector width is limited by the number of sites that evolve according to the same model, which corresponds to the length of the given partition.

Moreover, it is also difficult to apply the above scheme to the, otherwise efficient, PSR model of rate heterogeneity [86]. Instead of integrating the likelihood over different rates, it assigns one rate category (out of typically 25) to each alignment site. This means that, there are at least 25 different P matrices and that Equation 2.10 can only be vectorized across sites that evolve according to the *same* P matrix (rate category). Hence, devising a generic vectorization scheme for the PSR model, is not straight-forward.

6.3 GPU Implementation

We now describe the design and implementation of GPU kernels for the key functions of the PLL (`newview()`, `evaluate()`, and `coreDerivative()`). The model of rate heterogeneity is the Γ model. These functions are described in detail in Section 2.7, and account for more than 95% of total execution time in likelihood-based phylogenetic inference programs [10, 93]. For these functions, the ancestral probability vectors are read as input and, in the case of `newview()`, an additional APV is written as output.

These APV access patterns have two important implications in the design of the GPU implementation.

The *first* design criterion is associated with improved GPU thread performance when threads access contiguous memory locations in global memory. To maximize global memory throughput in the OpenCL model, it is essential to optimize memory coalescence and minimize address scatter [1]. The standard layout for APVs (see Figure 6.1) is problematic, because contiguous memory locations do not store likelihood entries belonging to the same state

and rate. We address this by using the ancestral probability vector memory layout (see Figure 6.2) presented in Section 6.2 and adapting it to GPUs.

The *second* challenge is that severe performance penalties are induced by frequently transferring large amounts of data between the CPU and GPU. Because the APVs dominate the memory requirements in the PLF computations (see discussion in Section 3.1), we devise an appropriate memory organization strategy. Under this specific scheme, the ancestral probability vectors are stored and updated exclusively on the GPU. The host program on the CPU simply orchestrates the tree search and invokes PLF computations on the ancestral probability vectors that reside on the GPU. We have developed an OpenCL kernel that implements this strategy. At each kernel call, the host program only needs to pass the memory addresses, that is, the starting positions of an ancestral probability vector (corresponding to a node of the tree) in GPU memory to the GPU. Apart from that, the CPU only needs to communicate the substantially smaller P matrices and one additional variable to the GPU. The variable indicates which PLF function (e.g., `newview()`, `evaluate()`, etc.) shall be executed.

Each kernel call returns at most one or two floating-point values to the CPU. In particular, calls to `evaluate()` return the overal log likelihood, and calls to `coreDerivative()` return the first and second derivatives of the likelihood function. When `newview()` is invoked, no values are returned because this function simply updates the ancestral probability vectors that reside in GPU memory.

6.3.1 Kernel Implementation

Update of ancestral probability vectors The CPU version of `newview()` (see Figure 6.4) includes three distinct cases: *tip-tip* (both children are leaves), *inner-tip* (one child is a leave and the other is an inner node), and *inner-inner* (both children are inner nodes). For details, see Section 2.8.

In the GPU version, these three cases have been reduced to one generic case (*inner inner*). This also induces a change in the layout of the vectors at the tips, which are stored in the form of inner ancestral probability vectors (*tip-APV*), rather than as a look-up table that is indexed by the raw alignment sequence data (for details see [89]).

While this doubles the memory requirements for storing ancestral probability vectors, it simplifies the storage of vectors in GPU memory, as well as the OpenCL code implementation (all cases are executed with the same kernel, avoiding conditional statements that would deteriorate performance). Hence, we allocate space for storing $2n - 2$ ancestral probability vectors on the GPU, where n is the number of taxa in the multiple sequence input align-

ment. Finally, the `newview()` function also implements a numerical scaling procedure to avoid numerical underflow in likelihood computations (for a detailed description, see [89]).

The GPU execution of the traversal descriptor described in Section 2.8 is depicted in Figure 6.4. Each entry in the descriptor represents an inner-inner operation, where three APVs are involved. The input APVs r and q are the child nodes. The ouput APV p is the parent of r and q. The parent/children relationship refers to the direction of the virtual root. The APVs r and q can be tip-APVs. The APV p always corresponds to an inner node in the tree. The branch length connecting the corresponding nodes of APVs p and q is given by b_{qp}. The branch length connecting p and r is given by b_{rp}. The traversal descriptor is processed sequentially as follows: for each entry in the descriptor, the host (CPU) computes the matrices $P(b_{rp})$ and $P(b_{qp})$. The host then transfers the APV identifiers of q, r and p (the APVs are stored in GPU global memory), and the P matrices to the device (GPU). The host then launches the `newview()` kernel. For each work-group, the corresponding P matrix is copied to local memory. Thus, threads belonging to the same work-group are executed in the same streaming multiprocessor, read and write contiguous memory positions (APV entries) from global memory, and have fast access to the P entries required to compute the new APV values. The kernel also performs additional operations related to numerical scaling. The host does not need to read back anything from the device.

Evaluation of the Likelihood The GPU implementation of `evaluate()` is analogous to the `newview()` kernel. Because of the changed ancestral probability vector representation at the tips, we simply omit the case where the left or right node of the branch at which the likelihood is calculated is a tip. The host must read back a double precision floating-point number (the log likelihood value).

Branch length optimization We observed that, during branch length optimization on the GPU, the overhead for invoking the pre-computation kernel (`sumGAMMA()`) and storing the results is larger than re-computing the product of the entries prior to each invocation of `coreDerivative()`. Hence, we merged `sumGAMMA()` and `coreDerivative()` into a single kernel.

Re-loading tip information in the GPU Apart from the $2n-2$ full ancestral probability vectors, we also store the raw alignment sequence data as well as the so-called `tipVector` data structure on the GPU. The reason for this is that we need to re-calculate the ancestral probability vectors at the

tips each time we change the values in the instantaneous substitution matrix Q (e.g., when optimizing the parameters of the General Time Reversible model of nucleotide substitution), which is required to calculate P. Changes to Q occur frequently when the rates in the Q matrix are being optimized.

While the values in the tip-APVs are normally expected to be constant, this is not the case for the numerical implementation of the PLF used in the PLL. In fact, the matrix of left Eigenvectors is multiplied with the tip probability vectors prior to any further likelihood calculations. This allows to save some computations later-on.

Each time the values in the Q matrix are changed this induces a change of the Eigenvector decomposition. As introduced in Section 2.1, the Eigenvector decomposition is used to exponentiate Q for obtaining the transition probability matrix P for a given branch length t (i.e., $P(t) = e^{Qt}$). Thus, the ancestral probability vectors at the tips need to be updated accordingly when the Eigenvector decomposition changes. However, transferring n tip vectors from the CPU to the GPU has a deteriorating effect on performance. Thus, we only transfer the substantially smaller `tipVector` array that contains the product of the left Eigenvectors with all possible DNA states (this is the look-up table used in the non-GPU PLL implementation) from the CPU to the GPU.

Once the `tipVector` is available on the device, the new ancestral probability vectors at the tips (tip-APVs) can be efficiently updated on the GPU using the already available raw alignment sequence data. The overhead of re-computing the ancestral probability vectors at the tips is negligible; it accounts for less than 1% of total run-time. In contrast to this, transferring all n ancestral probability vectors for the tips from the CPU to the GPU for each change in Q, induced a run time overhead of up to 70%.

6.3.2 GPU Memory Organization

Due to the aforementioned performance considerations, the APVs must be stored on the GPU. In order to correctly calculate the PLF, however, there are other APV-related data structures that must be stored in the GPU. We list these requirements below:

- Ancestral probability vectors for the $n-2$ inner nodes and n tips.
- The raw sequence alignment data of the tips that is required for re-computing the ancestral probability vectors at the tips when the Q matrix changes.

- The `tipVector` array that is also required for re-assembling the tip-APVs.
- A weight vector that indicates how many times each alignment site pattern occurs in the uncompressed MSA. This is called site pattern compression and is used in all standard PLF implementations to avoid unrequired computations (the per-site-LH is the same for two identical sites).
- Two `diagptable` arrays (one per child node) representing the P matrices for `newview()` and `evaluate()` invocations.
- A `globalScaler` array of size $2n-2$ for storing (and later-on un-doing) the scaling multiplications conducted to avoid numerical underflow at each node of the tree.
- Buffers to accumulate results, sum the per-site log likelihoods, and sum over the number of scaling multiplications.

In terms of memory requirements, this scheme is dominated by the $2n-2$ ancestral probability vectors. Thus, memory requirements can be approximated in advance. The proof-of-concept GPU implementation assumes that enough memory is available on the GPU. When this is not the case, it should be possible to apply the memory reduction strategy presented in Section 3.3, trading the lack of memory for additional computations. Alternatively, the computations can be split up among several GPUs.

6.3.3 OpenCL Implementation

In OpenCL (see Section 5.3), the work-group size, which is set by the kernel programmer, corresponds to the number of threads that are executed per streaming multiprocessor (SM). After experimenting with several multiples of 32, we empirically determined that a value of 64 worked best for our target application and HW platform. At each kernel call, all threads within a block can read data from contiguous positions in global memory. Thus, in our configuration we access 64 contiguous entries that share the same state and are evolving according to the same discrete Γ rate category. Our GPU kernel initializes the tips, and reads/writes the ancestral probability vectors according to this data layout ($workgroup := 64$), which is represented in Figure 6.3.

In order to improve performance, we applied optimization techniques such as loop unrolling [1], and storing the transition probabilities matrices and the

Node v							
Site $0/1/\dots/x$ interleaved							
Rate 0				Rate 1			
$L_{1,A}\dots L_{x,A}$	$L_{1,C}\dots L_{x,C}$	$L_{1,G}\dots L_{x,G}$	$L_{1,T}\dots L_{x,T}$	$L_{1,A}\dots L_{x,A}$	$L_{1,C}\dots L_{x,C}$	$L_{1,G}\dots L_{x,G}$	$L_{1,T}\dots L_{x,T}$

⋮

Node v							
Site $x+1/x+2/\dots/n$ interleaved							
Rate 0				Rate 1			
$L_{1,A}\dots L_{x,A}$	$L_{1,C}\dots L_{x,C}$	$L_{1,G}\dots L_{x,G}$	$L_{1,T}\dots L_{x,T}$	$L_{1,A}\dots L_{x,A}$	$L_{1,C}\dots L_{x,C}$	$L_{1,G}\dots L_{x,G}$	$L_{1,T}\dots L_{x,T}$

Figure 6.3: GPU Memory layout of an ancestral probability vector with work-group size x. Only 2 discrete Γ rates are represented.

eigenvectors in shared memory (local memory for each SM). We also explicitly use registers to store global variables that are read and written several times during kernel execution. The `newview` implementation is depicted in Figure 6.4.

6.4 Experimental setup and results

We simulated DNA data sets of different dimensions using INDELible [26] on random trees under the Jukes-Cantor model. Initially, we used INDELible (v1.03) to generate a large alignment of 15 taxa (species) and 900,000 sites. We then used a ruby script to extract subsets of unique site patterns from this alignment such as to generate 15-taxon datasets with distinct numbers of unique sites. Thus, in these datasets, the number of sites actually corresponds to the number of distinct alignment patterns which facilitates the discussion of the results. The use of simulated data is sufficient for measuring the performance of the PLL GPU version.

We executed the default RAxML-Light [91] search algorithm (version 1.0.5) to infer trees on these datasets. To conduct a realistic performance assessment for a real application using a GPU implementation of the PLF for the PLL, we re-implemented the search algorithm of RAxML-Light [91] using the PLL library. We compared the GPU implementation (see Section 6.3) and the AVX implementation based on the new generic memory layout (see Section 6.2) against the fastest serial version of RAxML-Light using the *ad hoc* AVX vectorization. The three code versions are implemented with the PLL library. They only differ with respect to their PLF implementations (`newview()`, `evaluate()`, `sumGAMMA()`, `coreDerivative()`). As mentioned before, the execution times for these functions dominate the run time. We manually instrumented the code to measure how much time is spent in each function. In the current datasets, the cumulative execution time of these

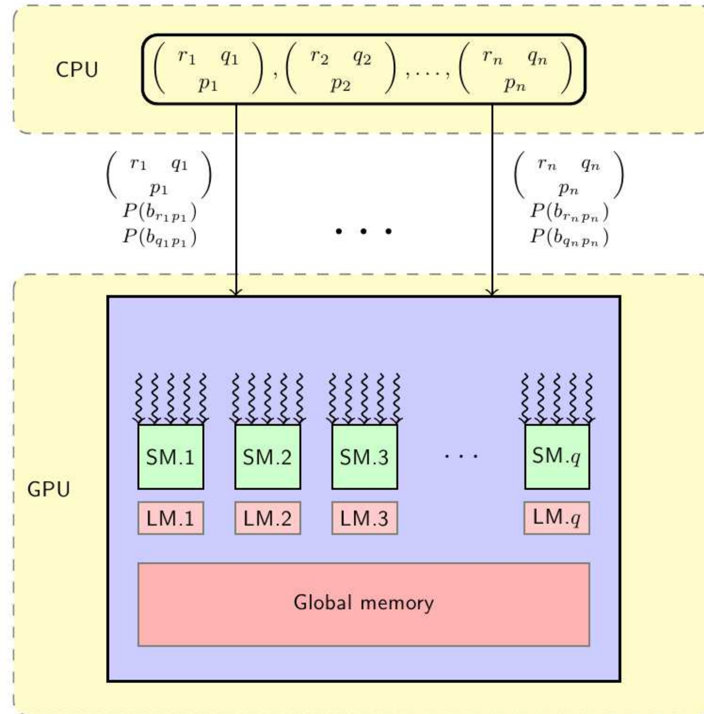

Figure 6.4: GPU implementation for `newview`. The CPU computes a traversal descriptor, which is a list of operations that must be executed before computing the likelihood at the virtual root. The host transfers the APVs identifiers of q, r and p, and the P matrices to the device (GPU). For each work-group, the corresponding P matrix is copied to local memory (LM). The APV entries for q and r are read from GPU global memory. The APV entries for p are computed on a SM, and written to global memory.

Patterns	2048	4096	8192	16384	32768	65536	131072	262144
AVX	**5.76**	**14.50**	**25.01**	**75.81**	117.30	300.05	508.45	1503.48
AVX-NEW	6.20	14.90	25.29	77.23	117.57	302.24	510.82	1506.45
GPU	40.19	52.75	50.12	90.83	**85.46**	**166.69**	**246.03**	**652.90**

Table 6.1: Total run times (in seconds) for the RAxML-Light search algorithm. AVX-NEW corresponds to the AVX implementation with the new generic memory layout for APVs.

four functions accounts for more than 95% of overall execution time on all datasets, and for 98% on the largest dataset with 262,144 unique site patterns.

The source code and results are available at `http://www.exelixis-lab.org/web/personal_page/izquierdo/gpu.tar.gz`

It is straight-forward to verify code correctness since the RAxML-Light search algorithm is deterministic for given, fixed starting trees. Thus, it is sufficient to compare the resulting tree topologies and log likelihood scores.

We executed the AVX versions (standard and new generic layout) on an Intel i5-3550 CPU running at 3.30GHz with 8GB RAM. The GPU version was executed on the same host system, which is also equipped with a NVIDIA Tesla C2075 card (448 CUDA cores, 1.15GHz, and 6GB GDDR5 of memory).

The total execution times are shown in Table 6.1. We achieved overall speedups exceeding a factor of two for the longest test dataset (see Figure 6.5). The three PLL kernels show comparable speedups. We observed a maximum speedup of three for the derivative computation. For `newview()`, which consumes the largest fraction of execution time, we obtained a maximum speedup of 2 (see Figure 6.6).

The number of scaling multiplications is proportional to the number of taxa (see [89] for details). To verify the correctness of the numerical scaling procedure, we also generated and executed a dataset with 200 taxa to force the codes to conduct numerical scaling. For this larger dataset, we did not run the full RAxML-Light algorithm, but a benchmark to evaluate the likelihood and optimize the branch lengths of a given a starting tree. As Table 6.2 shows, the scaling does not significantly affect GPU performance, that is, the speedups are comparable to those observed on the 15-taxa dataset, where scaling is not required.

Overall, there are no significant GPU speedups for DNA data. This is mainly because we are comparing the GPU code to the probably fastest currently available PLF implementation that relies on code that has been manually vectorized and tuned for AVX intrinsics. Thus, the GPU speedups obtained for DNA data are substantially lower than those reported in the

Patterns	2048	4096	8192
AVX	**2.04**	**4.13**	8.25
GPU	4.76	5.46	**7.66**

Table 6.2: Total run times (in seconds) for 10 evaluations and branch length optimization of a 200 taxa tree.

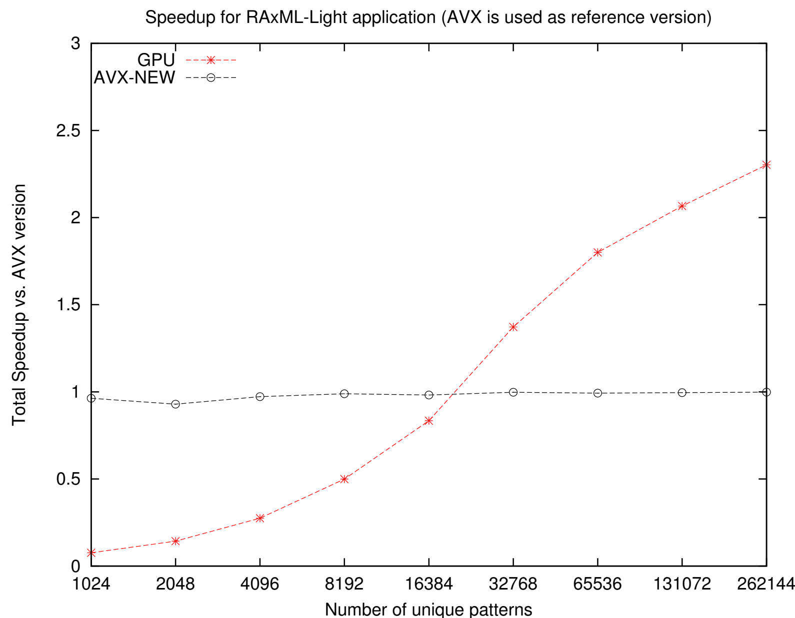

Figure 6.5: Speedups for a full application run of RAxML-Light. The reference is the AVX version with the standard layout.

BEAGLE paper. However, the speedups reported for BEAGLE compare GPU performance of BEAST, MrBayes, and GARLI, to plain C and SSE3-based implementations. Nonetheless, the performance of our PLL GPU implementation is expected to improve when models/data with more states are used, such as protein models, because they perform more computations per data accesses than for DNA data. Note that, the amount of PLF floating point computations per site increases with the squared number of states.

Nonetheless, GPUs can yield two-fold speedups over our highly optimized manual AVX implementation. Another interesting observation is that almost no performance penalty is induced by using the more generic vectorization scheme with AVX instructions.

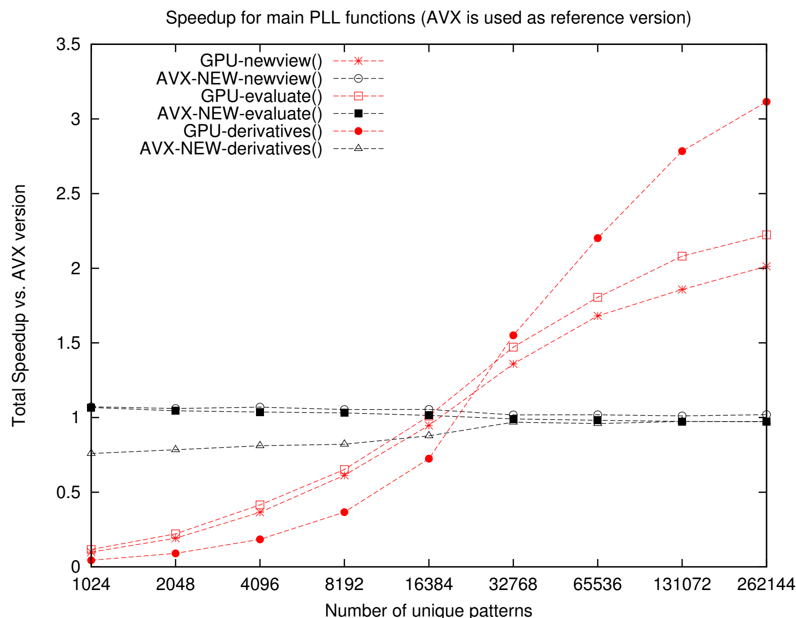

Figure 6.6: Speedups for each function of the PLL. The reference is the AVX version with the standard layout.

6.5 Summary

We have presented a GPU implementation for the main functions that are required for Bayesian and ML-based phylogenetic inference. In our approach, we store all ancestral probability vectors in the GPU memory to avoid transferring large amounts of data between the GPU and the CPU. We have also introduced an alternative and more generic layout for the ancestral probability vectors, which is suitable for x64 vector intrinsics and GPU architectures. This layout facilitates porting the library to larger x86 vector units that have recently become available.

Chapter 7

Perpetual Phylogenies with PUmPER

This chapter describes a framework named `PUmPER`, which makes use of the following publicly available open source tools: `PHLAWD` [81] (developed by Stephen A. Smith, who also implemented the features described in Subsection 7.2.1), RAxML-Light [91], `Parsimonator` [85], and standard-RAxML [87] (developed by Alexandros Stamatakis). John Cazes implemented the scripts described in Subsection 7.3.3. The content of this Chapter has been derived from the following peer-reviewed publication:
F. Izquierdo-Carrasco, J. Cazes, S. Smith, and A. Stamatakis. PUmPER: Phylogenies Updated Perpetually. *Bioinformatics*, 30(10):1476–1477, 2014

In this chapter we focus on topics related to automated inference and extension of phylogenetic trees. While the previous chapters focused on low level optimizations, here we discuss the benefits of automation with a focus on saving man-hours.

Existing phylogenies of taxonomic groups need to be updated as new data for new species, individuals and/or new genes are added to databases. The straight forward approach is to re-initiating phylogenetic inferences from scratch (every time data are added to public databases), which represents a waste of effort (man hours) and computations/energy. Nonetheless, adding new taxa or genes to a phylogenetic tree may also unravel new evolutionary relationships that were not supported by previous, smaller datasets. Albeit still an on-going debate, the taxon sampling density *can* have an impact on final tree shapes [9, 117].

Thus, it is worth exploring the tree topologies generated by datasets whose taxon sampling has been extended. In this chapter, we address these challenges, and present a framework called PUmPER (Phylogenies Updated Perpetually). PUmPER can iteratively construct multi-gene alignments (with PHLAWD) and phylogenetic trees (with RAxML-Light) for a given NCBI taxonomic group. Existing large reference phylogenies (and alignments) can be extended with PUmPER, without human intervention, and without the need to re-compute everything from scratch.

According to its configuration, PUmPER can detect when a sufficient amount of new data for the target clade and genes are available in GenBank, and automatically update existing alignments and phylogenies. We call this procedure a perpetual tree update. This can be helpful for commonly used datasets such as *rbc*L for seed plants, which has been used for broad scale phylogenies since 1993 [13], and also large ribosomal datasets that are used extensively [68].

PUmPER can be deployed either as a stand-alone tool on single machines, or on High Performance Computing systems, where phylogenetic inferences can be offloaded to a cluster. PUmPER is available (under the GNU GPL license) at `https://github.com/fizquierdo/perpetually-updated-trees`.
PUmPER does not only allow for extending phylogenies at a lower cost (in terms of energy and man hours), but it also yields equally good likelihood trees as *de novo* tree inferences conducted from scratch.

The rest of this chapter is organized as follows. In Section 7.1, we discuss previous work on automated inference and extension of phylogenies. In Section 7.2 we provide an overview of the PUmPER framework. Specific details about the software design of the stand-alone and distributed versions are provided in Section 7.3, including a description of the pipeline setup that perpetually updates the *Viridiplantae* clade. An evaluation of this pipeline is presented in Section 7.4.

7.1 Related Work

Previous work on perpetually updated trees focused on a framework called *mor*, which was designed for maintaining a specific automated taxonomy of *Homobasidiomycetes* [32]. The *mor* framework retrieves, screens, aligns, and analyzes nuc-lsu rDNA (nuclear large subunit ribosomal DNA) sequences of *Homobasidiomycetes* from GenBank and generates three phylogenetic trees each week. It generates an unconstrained jackknife neighbor-joining tree, a topologically constrained maximum parsimony consensus tree, and a topologically constrained maximum likelihood tree. While *mor* continuously updates

phylogenies, its main purpose is to produce automated taxonomies. To this end, a 'clade parser' is used to translate trees into rank-free classifications using node-based phylogenetic taxon definitions. For details, see [32]. The *mor* framework is accessible as a web service, where phylogenetic trees are published and archived on a weekly basis (at `http://mor.clarku.edu`). The *mor* software is written in `Perl` and available as open-source code. It can, in principle, be modified to work with any gene and group of organisms. At present, however, the web service is not being actively development anymore (pers. comm. with David Hibbett; March 27, 2013). The last archived files date back to 2008. The latest sequence added dates back to 2010, and the largest Maximum Likelihood tree produced by the system comprises 8019 taxa.

Further work has been done on automating the process of phylogenetic inference. STAP (Small Subunit rRNA Taxonomy and Alignment Pipeline) [108] is a pipeline that uses publicly available packages, such as ClustalW [103], PhyML [29] and BLASTN [3], to automate the process of phylogenetic inference and taxonomic assignment for ss-rRNA sequence data. STAP retrieves data from two public databases of ss-rRNA sequences: Greengenes [16] and RDP-II [49]. STAP is a collection of scripts available as a `Perl` package.

Phylometrics [84] is another automated pipeline for inference of phylogenetic trees. Like STAP, it uses BLASTN hits and ClustalW for alignment building, and PhyML for Maximum Likelihood tree inference. The pipeline is implemented in PHP as a web application, which can be installed locally or hosted remotely. In addition, batch jobs can be queued for each stage of the pipeline.

The total running time and memory usage of such automated pipelines is depends on the performance of their core components (e.g., ClustalW for alignment and PhyML for tree inference). They can also be compared in terms of feature availability, extendability, and usability. In this chapter, we present a flexible framework which can be used to automate the process of alignment construction and tree inference for arbitrary taxonomic groups. It can be deployed on both standalone (single machines) and High Performance Computing systems, enabling long-term availability of up-to-date phylogenetic trees.

7.2 Framework Overview

In the following, we outline the structure of `PUmPER`, our framework for perpetually updating phylogenetic trees of more than 20,000 taxa.

`PUmPER` is composed by (i) the multiple sequence alignment (MSA) gen-

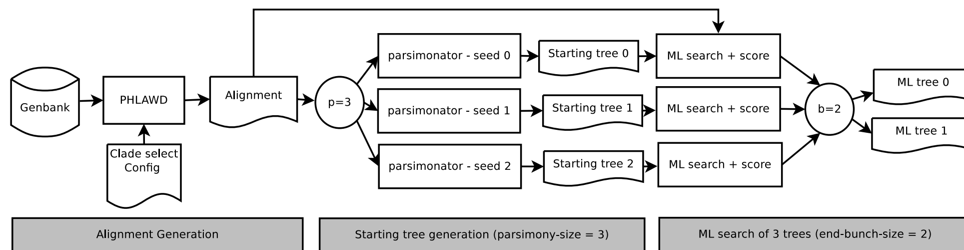

Figure 7.1: Initial iteration. An initial alignment is built for sequences covering a given clade and description search term (gene). Parsimony starting trees are used for Maximum Likelihood searches. The best trees are kept.

eration/extension component, and (ii) the phylogenetic tree inference component, which infers/extends the trees via Maximum Likelihood (ML) tree searches. The MSA component is an extension of `PHLAWD` [81], which can retrieve GenBank sequences and subsequently build or extend MSAs. The tree inference component is based on RAxML-Light [91], a dedicated HPC version of RAxML that can be executed on clusters using the Message Passing Interface (MPI). It can be used to infer new trees from scratch or to extend given trees by inserting additional taxa.

On top of these two components, we developed an iterative procedure that perpetually updates trees. Each iteration consists of two stages: the generation of a MSA, and the subsequent inference of a set of trees based on the generated MSA.

The *initial iteration* is special, since it builds the initial MSA and ML tree set from scratch. We call the remaining iterations *update iterations*, because they simply extend the MSAs and trees of the preceding iteration.

The setup of the initial iteration (see Figure 7.1) involves editing a configuration file for `PHLAWD`. In this file, the user must provide the NCBI taxonomic rank (clade name) and the gene(s) for which a MSA shall be assembled. `PUmPER` invokes `PHLAWD` to query GenBank and construct a initial MSA.

`PUmPER` uses `Parsimonator` to generate an initial set of distinct (randomized step wise addition order) parsimony starting trees based on this initial MSA. For each parsimony tree, `PUmPER` conducts an independent ML tree search with RAxML-Light to generate a set of ML trees. The user can specify the number of tree searches to be conducted and the size of the tree set to be kept in a configuration file.

In an update iteration (see Figure 7.2), `PUmPER` carries out the following four steps:

1. Update (re-assembly) of the MSA with `PHLAWD` with new GenBank data according to the initial configuration file.

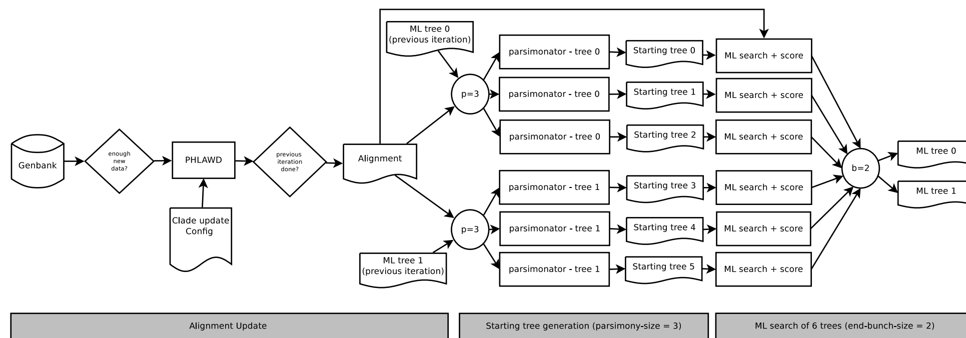

Figure 7.2: Update iteration. The alignment is extended and previous trees are re-used to continue searching in ML space with a set of different starting trees.

2. Generation of distinct randomized stepwise addition order parsimony starting trees with `Parsimonator` to extend the set of trees from the previous iteration with the newly added taxa.

3. ML optimization of the comprehensive parsimony starting trees (from step 2) with RAxML-Light.

4. Selection of a subset of these ML trees (based on their likelihood scores) that will be used as starting points for the next iteration.

7.2.1 MSA Construction/Extension with PHLAWD

MSA construction in `PHLAWD` has been described in [81], but for the purposes of understanding the perpetual procedure, we briefly outline the basic `PHLAWD` procedure again. `PHLAWD` requires the user to supply a configuration file that specifies for which organisms (as defined by the NCBI taxonomy) and which gene region(s) to construct a dataset. The user can identify the focal gene region by supplying a set of exemplar sequences. These sequences will be used for pairwise alignments and homology assessment. Additionally, the user can provide search terms that will be compared against the description of the sequences in the database to limit the scope of the sequence search. These are used in a Smith–Waterman procedure that discards sequences that are too dissimilar to the exemplars. Using the remaining sequences, `PHLAWD` attempts to construct MSAs. If the sequences included in the MSA are too divergent to construct a reliable MSA, `PHLAWD` splits up these sequences based on a guide tree, the default of which is the NCBI taxonomy. These subsets are initially aligned independently (with MAFFT [42]), then profile

alignment with MUSCLE [19] is used to align the subsets using the guide tree.

The PUmPER framework requires some adaptations in PHLAWD. We implemented an option to also pass user-supplied sequences to PHLAWD in addition to retrieving them from GenBank. This facilitated tests with simulated data, and also allows the user to extend the sequence set, by sequences that are not available in GenBank (e.g., simulated data or unpublished sequences). For tests with simulated data, we also extended PHLAWD to read-in user-supplied comprehensive (containing all taxa) or non-comprehensive (containing a subset of taxa) MSA guide trees. These trees can be used as an alternative to the NCBI taxonomy for splitting up sets of sequences for profile alignment and putting them back together. This feature can also be used to assist in the profile alignments of the user-supplied sequences. We also changed the underlying PHLAWD file organization. Previously PHLAWD stored all the intermediate alignments and other information in flat files. Now all files produced by a PHLAWD run are stored in a SQLite database file. This allows for easier replication of the PHLAWD MSA procedure by storing the order of subset alignment profiling. This is required for PHLAWD-based MSA extension when new sequences are added. The database also stores sequences that have been retrieved from GenBank and included in the MSA as well as sequences that have been added by users. We extended PHLAWD to allow for updating this *local* database with new sequences.

When PHLAWD has already been executed once and new sequences were added to the database (automatically or by the user), PHLAWD can be executed in update mode (updatedb option). Initially, the sequences of each new taxon are aligned to the closest existing subalignment. Depending on the information available, the closest subalignment is either determined by taxonomy or sequence similarity. Then, PHLAWD executes profile-profile alignments in the same order as in the original run. The information about the sub-alignment profile-profile alignment order is stored in the SQLite database. If the new MSA comprises 20% or more species than the preceding alignment, PHLAWD will re-divide the sequences into subsets and re-align them from scratch. The 20% threshold was empirically determined as 'good' default value but can be changed by the user.

Automated Assembly of Multi-gene datasets with PHLAWD

PUmPER also supports the generation of multi-gene alignments. For each gene region of interest, an independent PHLAWD instance (single-gene MSA) is run. Each instance has its own configuration and exemplar sequence file. Thereafter, PUmPER concatenates all single-gene MSAs into a multi-partitioned

dataset and stores the gene boundaries in a RAxML-formatted partition file. During an update iteration, each `PHLAWD` instance is extended independently (as described above).

The end result of a `PHLAWD` stage, be it initial or update, is a supermatrix stored in *PHYLIP* format in a folder for each iteration, which acts as an interface with the phylogenetic inference stage. Since these two stages are decoupled, it is straight-forward to substitute `PHLAWD` by another MSA construction method, that is, user-provided alignments can be used seamlessly with `PUmPER`.

7.2.2 Phylogenetic Inference

The second stage of every iteration is the phylogenetic inference of a set of trees based on the most recent MSA. The number of independent tree searches conducted at each iteration depends on two user parameters: the parsimony parameter p, and the size of the tree set b that shall be selected and kept in the end. In our experiments, we used constant numbers for p and b over all iterations ($p := 30$, $b := 10$ for initial iterations and $p := 3$, $b := 10$ for update iterations). These parameters can be modified by the user for each individual iteration. We denote the values of p and b for iteration i as $p(i)$ and $b(i)$.

Generation of starting trees

In the initial iteration $i := 0$, p determines how many randomized stepwise addition order parsimony starting trees will be generated (i.e., `Parsimonator` is invoked with p distinct random seeds).

In an update iteration $i > 0$, given an extended MSA and a set of selected ML trees (from the preceding iteration), $p(i)$ denotes how many comprehensive randomized stepwise addition order parsimony starting trees will be generated from the tree set of iteration $i-1$. Thus, `PUmPER` calls `Parsimonator` to extend the $b(i-1)$ trees from the preceding iteration. In each call, $p(i)$ distinct comprehensive parsimony trees are generated from the same preceding (non-comprehensive) ML tree.

Maximum Likelihood Inference

Each comprehensive parsimony starting tree topology is then optimized under ML with RAxML-Light. Thus, `PUmPER` conducts $p(0)$ ML searches for the initial iteration ($i = 0$) and $p(i) \cdot b(i-1)$ ML searches for all consecutive

iterations ($i > 0$). The choice of which flavor of RAxML-Light will be deployed (SSE3/AVX vectorization, Pthreads or MPI) depends on the available hardware.

Scoring and selecting the best trees

`PUmPER` waits until all ML searches have finished, and then scores the $p(i) \cdot b(i-1)$ topologies (`-f J` option) with standard-RAxML [87] under the Γ model of rate heterogeneity [110]. This option will also compute SH-like branch support values as described in [29]. Then, `PUmPER` selects the $b(i)$ best-scoring ML tree topologies, which will be the starting tree set used in iteration $i+1$, and finalizes the iteration.

7.2.3 Manual and automatic tree updates

Update iterations can be initiated manually through the command line interface. However, these updates can also be triggered automatically. An update iteration is started if (i) the alignment from the previous iteration has been extended *and* (ii) the phylogenetic analyses of the previous iteration have been completed. `PUmPER` can generate a `cron` job that periodically checks if the two conditions are true.

The MSA extension using `PHLAWD` can be automated via another `cron` job that periodically (default: once per week) queries GenBank and will launch `PHLAWD` to extend the MSA if enough new sequences (according to the configuration) have become available.

7.3 Software and Availability

`PUmPER` is available as open source code. In terms of installation requirements, all components used in the framework are open source and publicly available at `http://phlawd.net` (PHLAWD) and `http://www.exelixis-lab.org/` (RAxML). The design comprises Ruby modules that can be included in Ruby scripts. Each Ruby module encapsulates some independent functionality, that is, the user does not need to be aware of the specific usage of the underlying tools.

For instance, in Code Sample 1 we show part of the class implementation that abstracts the PHLAWD usage. The hash `@opts` contains a list of parameters read from user input and configuration files, and define the configuration of PHLAWD, as well as the database to be used.

Ruby Code Sample 1 Code extract from the PHLAWD wrapper. Multiple instances of PHLAWD are generated to independently build gene multiple sequence alignments.

```
class Phlawd
  def initialize(opts, log)
    @opts = opts
    @phlawd_runner = PhlawdRunner.new(log, @opts['phlawd_binary'])
    @instances = find_folder_instances
    @genbank_db = GenbankDB.new(@instances, @opts)
  end
  def run_initial
    @phlawd_iteration = PhlawdIteration.new(@phlawd_runner)
    # Run phlawd sequentially
    valid_instances.each do |instance|
      instance.run_initial unless File.exist? instance.result_file
      @phlawd_iteration.add_fasta_alignment instance
    end
    @phlawd_iteration
  end
  def find_folder_instances
    working_dir = @opts['phlawd_working_dir']
    instances = []
    Dir.entries(working_dir).reject{|f| f=~ /^\.+$/}.each do |f|
      path = File.join working_dir, f
      if File.directory? path
         instances << PhlawdInstance.new(path, @phlawd_runner)
      end
    end
    instances
  end
end
```

The application programmer can generate multi-gene alignments with a simple call as shown in Code Sample 2. Further detailed examples and configuration details are available in the code repository.

Ruby Code Sample 2 Calling PHLAWD from the PUmPER framework. A multi-gene dataset can be generated with automated subsequent calls to PHLAWD.

```
opts   = PerpetualTreeConfiguration::Configurator.new(ARGV.first)
log    = PerpetualTreeUtils::MultiLogger.new

phlawd = PerpetualPhlawd::Phlawd.new(opts, log)
msa    = phlawd.run_initial
```

Configuration files are used to determine specific settings. While our main use case is the automated update of phylogenetic trees, the framework can be easily used to build custom phylogenetic pipelines. For example, if alignments are already available, the `PHLAWD` component can be omitted. The on-line repository includes an installation guide, as well as basic usage and configuration examples.

7.3.1 Standalone implementation

When using the default configuration, `PUmPER` operates in stand-alone mode on a single server. `PHLAWD` and RAxML-Light are executed locally on this server. The distinct RAxML-Light tree searches are conducted one after the other, but the framework can be configured such that the Pthreads version of RAxML-Light is used on a multi-core machine. RAxML-Light implements the memory saving techniques described in Section 3.3 and Section 3.4. Thus, this stand-alone version allows for updating large trees on a medium-sized lab server.

7.3.2 Distributed implementation

For large trees, the computational resources of a single server may be insufficient due to memory and/or time constraints. Thus, `PUmPER` can also offload the computationally intense ML calculations to a cluster system. Thereby, the tree can be updated in a timely manner while the process is still orchestrated on a local server. This requires `PUmPER` to interface with remote systems using standard communications tools (`scp` and `ssh`), batch schedulers (`SGE`, `SLURM`, etc.), and to also use executables that have been optimized

for the remote system (`Parsimonator`, `RAxML-Light`, and `RAxML`). Although this adds another level of complexity, it is required for trees that take days and/or multiple nodes to process.

In the `PUmPER` framework, the local server orchestrates the remote work flow (ML calculations). By using cluster-specific configuration files and batch scheduler templates, the local server creates and submits batch scripts to execute steps 2, 3, and 4 of the update process (see Section 7.2). At the end of each step, the batch job transfers the results back to the local system.

The ML optimization of Step 3 may run for a long time and require (multiple) restarts from a checkpoint file. RAxML-Light offers such a checkpoint and restart facility, which allows for conducting a single tree inference in multiple steps if the run time exceeds the queue limits. For instance, for the setup described in Subsection 7.3.3, the standard queue time limit was 24 hours, which was occasionally exceeded, thus requiring a restart.

We have successfully used `PUmPER` with two popular job submission engines: `SGE` and `SLURM`. It should be straight-forward to adapt the current template files to other schedulers. However, cluster setups, security policies, queuing system configurations, etc., are different on each individual installation. Therefore, deploying `PUmPER` in conjunction with a cluster using `SGE` or `SLURM` might still need manual reconfiguration.

7.3.3 Custom iPlant setup

We are currently running a pipeline based on `PUmPER` as part of the iPlant collaborative (`http://www.iplantcollaborative.org/`). The goal is to maintain and make available perpetually updated trees for the *Viridiplantae* clade (using the *rbc*L, *matK*, and *atpB* genes). In this Section, we describe some details concerning the setup of this pipeline.

We use a dedicated server to control the workflow on a remote cluster. This server, Wooster, is a dedicated virtual machine (VM) provided by the iPlant collaborative to orchestrate the inference of perpetually-updated trees. It is configured with 8 Intel Westmere cores and 16 GB of memory. The processes running on the local server are relatively lightweight.

There were two clusters available, both at the Texas Advanced Computing Center (TACC), and part of the XSEDE (Extreme Science and Engineering Discovery Environment) program. The Linux cluster, *Lonestar*, was used during the development phase of the cluster computing component. The Lonestar cluster is composed of just under 2,000 compute nodes each with two 6-core Intel Westmere processors and 24 GB of RAM. Lonestar uses the Sun Grid Engine (SGE) batch facility to schedule jobs and allows users to connect directly via `ssh`. This batch scheduling facility along with the use of `ssh` for

remote commands allows the use of a locally managed master server, in this case Wooster, to distribute the computationally intensive tasks. By using template files to describe the cluster and the appropriate batch system, we have developed the cluster component to be portable to most HPC systems.

In January of 2013, a newer, more powerful cluster, Stampede, was made available at TACC, and is available to iPlant via an XSEDE project. Stampede is composed of 6,400 nodes, each with two 8-core Intel Sandy Bridge processors, 32 GB of RAM and a Xeon Phi Coprocessor. Although, RAxML, RAxML-Light, and `Parsimonator`, do not yet take advantage of the Xeon Phi, the RAxML family of codes contains AVX optimizations to take advantage of Sandy Bridge processors. Stampede differs from Lonestar in a few areas that require changes to the template file.

Since the cluster file and batch template files were developed for use on Lonestar and the SGE system, they had to be modified to run on the newer system, Stampede. This is easily done for the cluster file, which contains a description of the compute node resources and the path to the appropriate binaries. Below is an example of a cluster template file for Stampede.

```
# Info about cluster
cores_per_node: 16
mem_per_node: 31000
#Project required for batch scheduler
project: TG-MCB110022
submission: slurm

# Installed binaries, absolute paths in remote machine
parsimonator: ~/remote/wooster/bin/parsimonator-AVX
raxmllight: ~/remote/wooster/bin/raxmlLight-AVX
raxmllight_MPI: ~/remote/wooster/bin/raxmlLight-MPI-AVX
raxmllight_pthreads: ~/remote/wooster/bin/raxmlLight-PTHREADS-AVX
raxmlHPC_pthreads: ~/remote/wooster/bin/raxmlHPC-PTHREADS-SSE3
```

Since most of the job control logic has been integrated into the local server rather than the scheduler, the batch script templates are simple and may be ported to most batch scheduling systems. The only part that must be ported are the batch directives. An example of the `SLURM` directives required to run the RAxML-Light component of `PUmPER` is given below.

```
#SBATCH -J raxmllight_<%=params[:exp_name_run_num]%>
#SBATCH -d singleton  # ensures that each job with this name
                      # will only run one at a time
```

```
#SBATCH -n <%=params[:num_tasks]%>
#SBATCH -p normal
#SBATCH -o raxmllight_<%=params[:exp_name_run_num]%>.o%j
#SBATCH -e raxmllight_<%=params[:exp_name_run_num]%>.o%j
#SBATCH -t 24:00:00
#SBATCH -A <%=params[:project]%>
```

The *params* variables are set by the local server to launch individually named jobs. The only requirement of the scheduler is that it should be able to handle a simple job dependency in which jobs of the same name are run consecutively. The logic of staging input files and starting or restarting the RAxML-Light component is handled in the body of the batch file and should not need to be changed from system to system.

In our TACC setup, at the end of each iteration the best tree is uploaded to the iPlant collaborative tree visualization system, which uses Phyloviewer (publicly available at `http://portnoy.iplantcollaborative.org/`) to create a tree visualization on the iPlant server, Portnoy, with a link to the latest tree.

For example, Figure 7.3 shows the best-scoring and most up-to-date tree from Table 7.2.

7.4 Evaluation and Results

We have tested and evaluated `PUmPER` with simulated and real biological datasets. For each experiment (e.g., different clade or gene), we executed several iterations. For each update iteration, we also executed a control run which we denote as *scratch iteration*. A scratch iteration behaves like an initial iteration, that is, it builds the MSA from scratch on all sequences. It also executes the same number of independent ML tree searches as the update iteration, but *without* using previous topologies. We used the CONSEL package [78] to statistically assess if update and scratch iterations yield topologies with significantly different likelihood scores.

7.4.1 Biological examples

We have constructed two biologically relevant datasets for testing the `PUmPER` approach. The first dataset consists of the *rbc*L gene region for the clade of land plants (*Embryophyta*). The second dataset consists of the 18S ribosomal region for the *Eukaryota* clade.

`PHLAWD` uses identity and coverage to determine what sequences are similar enough based on Smith–Waterman comparisons. Values of 0.5 were used

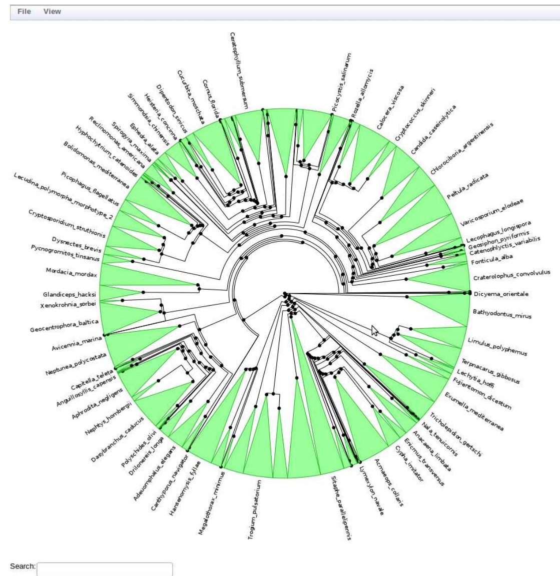

Figure 7.3: A perpetually updated tree for the 18S gene. Viewed with Phyloviewer (`http://portnoy.iplantcollaborative.org`).

for *rbc*L and 0.4 for 18S. These numbers were based on plots of identity and coverage using the `seqquery` command in `PHLAWD`. In order to simulate the update procedure on real data, each of these two datasets was created for sequences available before January 2008, before January 2010, and before September 2012.

Iteration	Taxa	Sites	Avg LH (30)	Avg LH (10)	Runtime(h)	Avg Branch support
2008	12072	1437	-848794.80	-848745.23	46.55	67.78
2010	16962	1427	-1005824.25	-1005762.81	68.36	64.25
2010 scratch	16962	1427	-1005931.37	-1005863.32	70.89	64.26
2012(Sept)	21791	1424	-1108161.66	-1107598.42	93.40	59.56
2012(Sept)scratch	21791	1424	-1108243.29	-1107774.80	97.42	59.46

Table 7.1: Original run and two updates of the *rbc*L datasets. Average ML scores at the end of each iteration (averaged over all 30 trees and the 10 best trees) and overall run times of all searches. The branch support is the average of SH-like support values on the best tree. The running time is the sum of the 30 ML searches.

Iteration	Taxa	Sites	Avg LH (30)	Avg LH (10)	Runtime(h)	Avg Branch support
2008	14634	2363	-1950468.79	-1950281.56	74.16	70.24
2010	18480	2214	-2340314.06	-2340142.38	88.73	68.73
2010 scratch	18480	2214	-2340563.92	-2340260.37	94.08	68.59
2012(Sept)	23298	2110	-2782234.35	-2781965.38	116.48	68.13
2012(Sept)scratch	23298	2110	-2782132.94	-2781959.11	124.23	68.00

Table 7.2: Original run and two updates of the 18S datasets. Average ML score at the end of each iteration (averaged over all 30 trees and the 10 best trees) and overall run time of all searches. The branch support is the average of SH-like support values on the best tree. The running time is the sum of the 30 ML searches.

Table 7.1 and Table 7.2 show the average ML scores for three update iterations. Each update iteration was run with parameters $p := 3$, $b := 10$, resulting in 30 independent ML searches.

We concatenated all resulting trees from *scratch* and *update* iterations based on the same biological real MSA, and used the CONSEL package [78] to assess the confidence of phylogenetic tree selection, that is, which trees were significantly better in terms of likelihood scores. The confidence set according to the approximately unbiased (AU) test [76], delimited by a p-value of $p := 0.05$, included for each year and dataset topologies from both the *scratch* and *update* iterations.

For example, in Table 7.3 we present the resulting confidence set of trees (out of all possible 60 trees) that were selected by CONSEL for the *18S* dataset update labelled as *2010* (see Table 7.2). Trees with ids from 1 to 30 correspond to the PUmPER trees, and trees with id from 31 to 60 correspond to the restart approach. We clearly see that these tree sets, which are not significantly different from each other, include trees from both approaches.

Rank	Tree id	LH difference	AU (p-value)
1	**18**	-26.5	0.701
2	**29**	26.5	0.666
3	**26**	36.6	0.672
4	**27**	71.5	0.602
5	41	125.7	0.517
6	**1**	133.9	0.504
7	60	140.6	0.476
8	40	145.5	0.464
9	**20**	167.4	0.448
10	59	173.3	0.443
11	52	183.8	0.462
12	36	185.6	0.423
13	**19**	199.8	0.317
14	**6**	203.8	0.358
15	**8**	230.0	0.308
16	**2**	234.3	0.343
17	**16**	271.5	0.270
18	**11**	279.4	0.274
19	**30**	320.0	0.201
20	**15**	320.6	0.284
21	**13**	345.0	0.253
22	**14**	345.2	0.178
23	**21**	356.5	0.222
24	**4**	375.2	0.177
25	39	413.6	0.189
26	**28**	428.8	0.122
27	**10**	429.0	0.111
28	57	430.0	0.171
29	56	440.3	0.203
30	**25**	441.7	0.105
31	55	456.8	0.138
32	**7**	458.5	0.138
33	**9**	460.3	0.049
34	51	470.2	0.095
35	**12**	471.0	0.103
36	**23**	517.3	0.077
37	**22**	528.8	0.052
38	42	545.7	0.034
39	45	558.8	0.082
40	32	562.1	0.057
41	37	616.9	0.042
42	47	620.1	0.057

Table 7.3: AU test for iteration 2010 of 18S dataset. Trees are ranked by log likelihood (first column). The second column shows the tree id, 1 to 30 are PUmPER trees (bold), 31 to 60 scratch trees. The third column shows the log LH difference with the best tree. The fourth column is the p-value as computed by the AU (aproximately unbiased) test. Only trees within the confidence set ($p > 0.05$) are shown.

7.4.2 Simulated Data

We used INDELible [26] to generate a simulated dataset with 9097 taxa on a tree inferred on the *rbc*L gene for the *Viridiplantae* clade. We used the simulated sequences and the underlying true tree as guide tree to asses the iterative MSA update procedure in `PHLAWD`.

From the 9079 existing sequences in the simulated alignment, we selected 4000 at random as user sequences to generate the initial `PHLAWD`-based MSA. The remaining sequences were randomly distributed across 3 update blocks of 1345, 1396, and 2329 sequences. Each update block was used as user-sequences to extend the MSA, generating extended MSAs of 5345, 6741, and 9079 taxa.

We pruned the true tree accordingly such that for each MSA and iteration a corresponding true tree was available. We used these pruned true trees to determine the topological accuracy of the inferred trees (at each iteration) using the Robinson-Foulds distance [69], denoted as RF-distance.

As before, each iteration included a total of 30 independent ML searches. We observed that starting from extended topologies (update iterations) does neither increase nor decrease topological accuracy (see Table 7.4).

Iteration	Taxa	Sites	Avg LH (30)	Avg LH (10)	RF (true tree)	Runtime(h)	Avg support
0	4000	1500	-589036.97	-589035.58	0.146	2.92	77.63
1	5345	1500	-715683.68	-715682.66	0.163	7.46	76.14
1 scratch	5345	1500	-715682.75	-715681.41	0.162	7.89	76.16
2	6741	1500	-838437.33	-838436.48	0.176	6.21	74.72
2 scratch	6741	1500	-838440.38	-838438.06	0.174	8.17	74.73
3	9079	1500	-1033499.23	-1033498.23	0.184	10.10	73.12
3 scratch	9079	1500	-1033498.15	-1033495.85	0.185	19.27	73.12

Table 7.4: Original run and three simulated updates of the simulated datasets. Average Likelihood at the end of each iteration (30 total trees and 10 best trees) and total run time of all searches. The branch support values are the average of SH-like support values of the best tree. The running time is the sum of the 30 maximum likelihood searches.

7.5 Discussion

According to our first results, the iterative MSA and tree extension (PUmPER) approach yields neither significantly better, nor worse trees than the standard (inference from scratch) approach with respect to the likelihood scores. The topological accuracy in our simulations is comparable in both approaches. The relatively high RF distances are expected for phylogenetic reconstructions on short simulated single-gene datasets [44].

The runtimes in the PUmPER (iteration-based) approach are slightly, but consistently lower. We view the main contribution, however, in saving man-hours: alignment construction, job setup, filtering, and post-analyzing results are tedious tasks that consume a significant, and hard to estimate, amount of time.

The framework offers the required flexibility to set up self-maintained on-going analysis such as, for example, simultaneous perpetual inference of gene/species trees using multi-gene and single-gene phylogenetic inferences.

7.6 Summary

We have presented and made available a framework named PUmPER that can be used to maintain and perpetually update phylogenetic trees. We have used this framework to implement and set up a pipeline for updating phylogenies based on multi-gene alignments for the *Viridiplantae* clade. PUmPER can operate in stand-alone mode on a single server, but also offload computationally expensive ML searches to an external cluster. The perpetually updated phylogenies can be computed slightly faster and are not significantly (in the statistical sense) worse nor better than phylogenies that are inferred from scratch.

We are currently using PUmPER to maintain a perpetually updated phylogeny for the green plants within the framework of the iPlant collaborative.

Chapter 8

Conclusion And Outlook

8.1 Conclusion

The main goal of this thesis was to study and develop methods for analysing large-scale datasets for Maximum Likelihood based phylogenetic inference. We have approached this challenge using different strategies: reduction of memory requirements, reduction of running time, and reduction of man-hours.

The reduction of memory footprints involved three different techniques. The out-of-core and the recomputation strategy share the idea of only storing a subset of the required ancestral probability vectors in memory. The remaining vectors are stored on disk (out-of-core) or obtained through additional computations (recomputation). Our assessment shows that the recomputation approach clearly outperforms the out-of-core strategy. We have proved that the recomputation technique can compute the log likelihood by storing only $log_2(n) + 2$ APVs for trees with n taxa. Furthermore, the overhead in running time is negligible for full tree traversals. For partial tree traversals, the overhead is typically 10% in running time when only half of the required APVs are stored in memory. These features can be useful for inference of large trees when memory is a limiting factor.

The re-implementation of the SEV technique can reduce memory almost proportionally to the amount of missing data, without sacrificing runtime. Furthermore, the SEV and recomputation techniques are orthogonal and can thus be deployed simultaneously, as we have shown in the RAxML-Light implementation.

We have also presented an algorithm to reduce the tree size and thus the space of possible tree topologies during tree search. We have shown that, for large topologies comprising tens of thousands of taxa, the resulting trees

can be computed faster and are not significantly worse than trees obtained from standard searches. Backbone-based likelihood scores where consistently lower, but correlated with topologies inferred with the same starting tree under a standard search. Therefore, we conclude that this approach can be useful to identify good starting trees. In other words, the backbone approach could be used to discard non-promising starting trees during the early phase of the phylogenetic search.

We have explored and efficiently ported the computation of the phylogenetic likelihood function to GPUs. This required adapting the memory layout for ancestral probability vectors in order to achieve optimal performance for a GPU architecture. We have shown that, for large datasets, offloading PLF computations on DNA data to the GPU can be up to two times faster than the most efficient CPU vectorized code for our RAxML benchmark code.

While we used the RAxML codebase to develop proof-of-concept implementations of the above mentioned techniques, they are generic enough to be applicable to other state-of-the-art likelihood-based codes.

Finally, we have released the `PUmPER` open source framework, which can be used to build perpetual phylogenetic pipelines. This framework can be used to maintain and extend up-to-date comprehensive reference trees containing all taxa of a specific taxonomic rank. `PUmPER` is written in Ruby and can be configured to operate in stand-alone mode on a single server, or to offload computationally expensive maximum likelihood searches to an external cluster. Currently, we are using this pipeline to maintain and update phylogenies for a multi-gene alignment of the *Viridiplantae* clade.

8.2 Future Work

Most of the potential future work concerns the GPU implementation and the `PUmPER` framework. The out-of-core approach has been completely abandoned since it was clearly outperformed by the recomputation approach. The other two orthogonal memory-saving techniques (SEV and recomputation) are currently being integrated in the codebase of the PLL library.

8.2.1 GPUs

With respect to future work on the GPU implementation of the PLF, we plan to fully integrate the proof-of-concept GPU kernel with the PLL and support all models and data types (e.g., protein data, partitioned datasets, and the PSR model of rate heterogeneity). Another possible enhancement is

to implement the re-computation technique described in Subsection 3.3.1 on GPUs, since this would enable the computation of larger datasets on GPUs.

In order to leverage the computational power of modern desktop systems and clusters, we intend to implement a hybrid CPU/GPU system (using GPUs and x86 cores). The underlying idea is that, during the execution of the GPU Kernel, the CPU does not need to wait for the kernel to complete, but that it can also execute some PLF computations. For instance, the APVs can be divided into disjoint fragments that are assigned to the GPU and to the CPU. When the operations stored in the traversal descriptor are executed, each processing unit can update asynchronously its APV fragment. Once all fragments have been updated, the CPU can easily combine the partial results when required, for instance, for `evaluate()` and `coreDerivative()`. While the CPU and GPU implementations are already available, the implementation of this approach is not straight-forward, because load balancing issues must be addressed.

8.2.2 PUmPER

Future developments for the `PUmPER` project include developing a web-service to facilitate the use of the automated update pipeline for a broader community. The framework modules can be easily adapted to fit a Model-View-Controller (MVC) based web application. In this context, the user could configure his perpetually updated phylogenetic inference (genes, clade, number of phylogenies) and visualize the resulting trees from a web browser.

We also intend to integrate the recently introduced approximation techniques for computing bootstrap values [54] into RAxML-Light and the framework, and consider replacing RAxML-Light by the newer and more efficient ExaML code [90] for phylogenetic inference.

8.3 Outlook

Given the current developments in Next-Generation Sequencing technologies, such as single molecule sequencing [58], the cost for sequencing genomes is expected to keep decreasing in the next years. At the same time, the scale of on-going and planned biological data analysis projects is larger than ever in terms of data volume. For instance, the 100K Genome Project `http://www.genomicsengland.co.uk/` will sequence the full genomes of 100,000 patients over the next five years.

In this context, the techniques presented here will become more relevant over the next years. In particular, we envision that phylogenetic inference of

alignments including hundreds or thousands of genes may become common practice. The GPU implementation, whose performance improves with sequence length, may become more efficient for these type of analyses. Thus, further work needs to be conducted to improve the performance of these and upcoming hardware architectures, as well as on the design of the aforementioned hybrid CPU/GPU system.

Furthermore, the grand challenge of inferring the tree of life, that is a phylogeny comprising all described species, remains unsolved. Inferring larger trees comprising more species is required to understand biological questions, such as species diversification [67, 80], that cannot be answered by analysing smaller phylogenies. The inference of such large-scale phylogenies, however, poses challenges related to the alignment size. On the one hand, memory requirements increase linearly with the sequence length and the number of species. These requirements can be reduced by simultaneously applying the orthogonal memory saving techniques (Subtree Equality and recomputation of ancestral vectors). On the other hand, adding more taxa to the alignment increases the amount of possible topologies in tree space. Thus, running time will remain a limiting factor, and exploring aggressive heuristics to reduce tree space may, therefore, be required. To this end, approaches like the backbone algorithm may be a good starting point for future research.

In general, the techniques presented in this thesis can help to improve the scalability of current state-of-the-art phylogenetic codes, and thus enable the analysis of large phylogenies at the genome scale.

List of Figures

2.1 From sequences to alignment 9
2.2 Concatenated multiple sequence alingment 10
2.3 Protein 12
2.4 Unrooted Phylogenetic Tree 13
2.5 Unrooted Phylogenetic Tree with branch lengths 13
2.6 Unrooted Phylogenetic Tree with a virtual root 17
2.7 Ancestral Probability Vectors 18
2.8 Density function for the Γ distribution 20
2.9 A lazy SPR move 22
2.10 Node records 26

3.1 Standard Memory Layout of an ancestral probability vector . 32
3.2 Vectors stored on memory and disk 36
3.3 Miss rates for different replacement strategies 40
3.4 Effect of Read skipping 41
3.5 Miss rates for fractions of memory 41
3.6 Execution times for full tree traversals 43
3.7 Recomputation with a Balanced Subtree 46
3.8 Recomputation with an unbalanaced subtree 52
3.9 Replacement Strategies 55
3.10 Overall RAM usage with partial allocation 56
3.11 Generic SEVs 59
3.12 SEV savings in computations 61
3.13 SEV savings in computations and memory 62

4.1 Consistency of labels at the backbone boundaries 72
4.2 Increase of backbone tips due to topology conflicts 73
4.3 Log Likelihood scores for the 38K dataset 75
4.4 Log Likelihood scores for the 56K dataset 76

5.1 The Fermi Architecture 81
5.2 Architecture of a streaming multiprocessor 81

5.3 CUDA memory hierarchy 83
5.4 Coalesced access to global memory 84

6.1 Standard Memory Layout 89
6.2 Memory Layout for vector width of 2 89
6.3 Memory Layout for vector in GPU 96
6.4 GPU implementation for newview() 97
6.5 GPU speedups (full run) 99
6.6 GPU speedups (functions) 100

7.1 Initial Iteration 104
7.2 Update Iteration 105
7.3 A perpetually updated tree for the 18S gene. 114

List of Tables

3.1 Frequency of ancestral vector cases for different strategies. . . . 57
3.2 Average run times for full traversals 57
3.3 SEV Evaluation for the 38K dataset 63
3.4 SEV Evaluation for the 56K dataset 63

4.1 Evaluation of methods to identify the innermost node 70
4.2 Average number of computed backbone tips 72
4.3 Average Performance for the 38K dataset 77
4.4 Average Performance for the 56K dataset 77
4.5 Average Symmetric Difference 78

6.1 GPU evaluation 98
6.2 GPU scaling evaluation 99

7.1 PUmPER iterations of the *rbc*L dataset 115
7.2 PUmPER iterations of the 18S dataset 116
7.3 AU test for iteration 2010 of 18S dataset 118
7.4 PUmPER iterations on simulated dataset 120

List of Acronyms

AA - Aminoacid
API - Application Programming Interface
APV - Ancestral Probability Vector
AS - Ancestral State
AVX - Advanced Vector Extensions
BS - Bootstrap Support
CPU - Central Processing Unit
CUDA - Compute Unified Device Architecture
DNA - Deoxyribonucleic acid
EM - External Memory
GPGPU - General Purpose computation on GPU
GPU - Graphics Processing Unit
GTR - General Time Reversible
LGT - Lateral Gene Transfer
NJ - Neighbour Joining
NNI - Nearest Neighbour Interexchange
OTU - Operational Taxonomic Unit
PLF - Phylogenetic Likelihood Function
PLL - Phylogenetic Likelihood Library
PSR - Per-Site Rate (model of rate heterogeneity)
RNA - Ribonucleic acid
SDK - Software Development Kit
SIMD - Single Instruction, Multiple Data
SIMT - Single Instruction, Multiple Thread
SPR - Subtree Pruning and Regrafting
SSE3 - Streaming SIMD Extensions 3
TBR - Tree Bisection and Reconnection
TU - Taxonomic Unit
VM - Virtual Machine
VW - Vector (Unit) Width

Bibliography

[1] N. Alachiotis, S. A. Berger, and A. Stamatakis. Coupling simd and simt architectures to boost performance of a phylogeny-aware alignment kernel. *BMC Bioinformatics*, 13:196, 2012.

[2] Altera. White paper implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform, 2007.

[3] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389, 1997.

[4] D. L. Ayres, A. Darling, D. J. Zwickl, P. Beerli, M. T. Holder, P. O. Lewis, J. P. Huelsenbeck, F. Ronquist, D. L. Swofford, M. P. Cummings, A. Rambaut, and M. A. Suchard. BEAGLE: An Application Programming Interface and High-Performance Computing Library for Statistical Phylogenetics. *Systematic Biology*, 61(1):170–173, 2012.

[5] E. Barkan, E. Biham, and A. Shamir. Rigorous bounds on cryptanalytic time/memory tradeoffs. In *In Advances in Cryptology-CRYPTO 2006, volume 4117 of LNCS*, pages 1–21. Springer-Verlag, 2006.

[6] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers. GenBank. *Nucleic acids research*, 37(Database issue):D26–31, Jan. 2009.

[7] S. A. Berger and A. Stamatakis. Accuracy and performance of single versus double precision arithmetics for maximum likelihood phylogeny reconstruction. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 6068 of *Lecture Notes in Computer Science*, pages 270–279. Springer Berlin Heidelberg, 2010.

[8] S. A. Berger and A. Stamatakis. PaPaRa 2.0: A Vectorized Algorithm for Probabilistic Phylogeny-Aware Alignment Extension; Exelixis-RRDR-2012-5; http://sco.h-its.org/exelixis/pubs/Exelixis-RRDR-2012-5.pdf. Technical report, Heidelberg Institute for Theoretical Studies, 2012.

[9] O. Bininda-Emdons and A. Stamatakis. *Reconstructing the Tree of Life: taxonomy and systematics of species rich taxa*, volume 72, chapter Taxon sampling versus computational complexity and their impact on obtaining the Tree of Life, pages 77–95. CRC press, 2006.

[10] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos. RAxML-Cell: Parallel Phylogenetic Tree Inference on the Cell Broadband Engine. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS2007)*, 2007.

[11] A. R. Brodtkorb, T. R. Hagen, and M. L. Saetra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, 73(1):4 – 13, 2013.

[12] M. Charalambous, P. Trancoso, and A. Stamatakis. Initial Experiences Porting a Bioinformatics Application to a Graphics Processor. In *Proc. of the 10th Panhellenic Conference on Informatics (PCI 2005)*, pages 415–425, 2005.

[13] M. W. Chase, D. E. Soltis, R. G. Olmstead, D. Morgan, D. H. Les, B. D. Mishler, M. R. Duvall, R. A. Price, H. G. Hills, Y.-L. Qiu, K. A. Kron, J. H. Rettig, E. Conti, J. D. Palmer, J. R. Manhart, K. J. Sytsma, H. J. Michaels, W. J. Kress, K. G. Karol, W. D. Clark, M. Hedren, B. S. Gaut, R. K. Jansen, K.-J. Kim, C. F. Wimpee, J. F. Smith, G. R. Furnier, S. H. Strauss, Q.-Y. Xiang, G. M. Plunkett, P. S. Soltis, S. M. Swensen, S. E. Williams, P. A. Gadek, C. J. Quinn, L. E. Eguiarte, E. Golenberg, J. Learn, Gerald H., S. W. Graham, S. C. H. Barrett, S. Dayanandan, and V. A. Albert. Phylogenetics of seed plants: An analysis of nucleotide sequences from the plastid gene rbcl. *Annals of the Missouri Botanical Garden*, 80(3):pp. 528–548+550–580, 1993.

[14] K. G. Consortium. Opencl: The open standard for parallel programming of heterogeneous systems.

[15] D. Darriba, A. Aberer, T. Flouri, T. 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 Workshops and Phd Forum (IPDPSW), 2013 IEEE International Symposium on, 2013.

[16] T. DeSantis, P. Hugenholtz, N. Larsen, M. Rojas, E. Brodie, K. Keller, T. Huber, D. Dalevi, P. Hu, and G. Andersen. Greengenes, a chimera-checked 16s rrna gene database and workbench compatible with arb. *Appl Environ Microbiol*, 72(7):5069–72, 2006.

[17] A. Drummond and A. Rambaut. Beast: Bayesian evolutionary analysis by sampling trees. *BMC evolutionary biology*, 7(1):214, 2007.

[18] P. Dri and Z. Galil. A time-space tradeoff for language recognition. *Theory of Computing Systems*, 17:3–12, 1984. 10.1007/BF01744430.

[19] R. C. Edgar. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, 2004.

[20] A. W. F. Edwards and L. L. Cavalli-Sforza. Reconstruction of evolutionary trees. *Systematics Association Publication.*, 6:67–76, 1964.

[21] I. Elias. Settling the intractability of multiple alignment. *J. Comput. Biol.*, 13(7):1323–1339, Sep 2006.

[22] J. Fang, A. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225, 2011.

[23] J. Felsenstein. Evolutionary trees from DNA sequences: a maximum likelihood approach. *J. Mol. Evol.*, 17:368–376, 1981.

[24] J. Felsenstein. Confidence Limits on Phylogenies: An Approach Using the Bootstrap. *Evolution*, 39(4):783–791, 1985.

[25] J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Inc., 2004.

[26] W. Fletcher and Z. Yang. INDELible: a flexible simulator of biological sequence evolution. *Molecular biology and evolution*, 26(8):1879–1888, 2009.

[27] T. Flouri, F. Izquierdo-Carrasco, A. Stamatakis, et al. PLL: Phylogenetic likelihood library; http://www.libpll.org. *Work in Progress*, 2013.

[28] P. A. Goloboff, S. A. Catalano, J. M. Mirande, C. A. Szumik, J. S. Arias, M. Källersjö, and J. S. Farris. Phylogenetic analysis of 73060 taxa corroborates major eukaryotic groups. *Cladistics*, 25:1–20, 2009.

[29] S. Guindon, J. Dufayard, V. Lefort, M. Anisimova, W. Hordijk, and O. Gascuel. New algorithms and methods to estimate maximum-likelihood phylogenies: assessing the performance of phyml 3.0. *Systematic biology*, 59(3):307, 2010.

[30] J. Hauser, K. Kobert, F. Izquierdo-Carrasco, K. Meusemann, B. Misof, M. Gertz, and A. Stamatakis. Heuristic algorithms for the protein model assignment problem. In Z. Cai, O. Eulenstein, D. Janies, and D. Schwartz, editors, *Bioinformatics Research and Applications*, volume 7875 of *Lecture Notes in Computer Science*, pages 137–148. Springer Berlin Heidelberg, 2013.

[31] T. Heath, M. Holder, and J. Huelsenbeck. A dirichlet process prior for estimating lineage-specific substitution rates. *Molecular biology and evolution*, 29(3):939–955, 2012.

[32] D. S. Hibbett, R. H. Nilsson, M. Snyder, M. Fonseca, J. Costanzo, and M. Shonfeld. Automated phylogenetic taxonomy: An example in the homobasidiomycetes (mushroom-forming fungi). *Systematic Biology*, 54(4):660–668, 2005.

[33] D. H. Huson, R. Rupp, and C. Scornavacca. *Phylogenetic Networks*. Cambridge University Press, Cambridge, 2010.

[34] L. Inc. Lockless memory allocator; http://locklessinc.com.

[35] F. Izquierdo-Carrasco, N. Alachiotis, S. Berger, T. Flouri, S. P. Pissis, and A. Stamatakis. A generic vectorization scheme and a gpu kernel for the phylogenetic likelihood library. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2013 IEEE International Symposium on*, 2013.

[36] F. Izquierdo-Carrasco, J. Cazes, S. Smith, and A. Stamatakis. PUmPER: Phylogenies Updated Perpetually. *Bioinformatics*, 30(10):1476–1477, 2014.

[37] F. Izquierdo-Carrasco, J. Gagneur, and A. Stamatakis. Trading running time for memory in phylogenetic likelihood computations. In J. Schier, C. M. B. A. Correia, A. L. N. Fred, and H. Gamboa, editors, *BIOINFORMATICS*, pages 86–95. SciTePress, 2012.

[38] F. Izquierdo-Carrasco, S. Smith, and A. Stamatakis. Algorithms, data structures, and numerics for likelihood-based phylogenetic inference of huge trees. *BMC Bioinformatics*, 12(1):470, 2011.

[39] F. Izquierdo-Carrasco and A. Stamatakis. Computing the phylogenetic likelihood function out-of-core. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 444–451, 2011.

[40] F. Izquierdo-Carrasco and A. Stamatakis. Inference of huge trees under maximum likelihood. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2490–2493. IEEE, 2012.

[41] T. Jukes and C. Cantor. *Evolution of protein molecules.*, chapter III, pages 21–132. Academic Press, New York, 1969.

[42] K. Katoh and H. Toh. Recent developments in the MAFFT multiple sequence alignment program. *Briefings in Bioinformatics*, 9(4):286–298, 2008.

[43] H. Kishino and M. Hasegawa. Evaluation of the maximum likelihood estimate of the evolutionary tree topologies from dna sequence data, and the branching order in hominoidea. *Journal of Molecular Evolution*, 29(2):170–179, 1989.

[44] A. Kupczok, H. Schmidt, and A. Haeseler. Accuracy of phylogeny reconstruction methods combining overlapping gene data sets. *Algorithms for Molecular Biology*, 5(1):1–17, 2010.

[45] C. Lanave, G. Preparata, C. Saccone, and G. Serio. A new method for calculating evolutionary substitution rates. *Journal of Molecular Evolution*, 20:86–93, 1984.

[46] N. Lartillot, S. Blanquart, and T. Lepage. PhyloBayes. v2. 3, 2007.

[47] D. Lipman and W. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[48] P. Li and N. Goldman. Models of molecular evolution and phylogeny. *Genome Research*, 8:1233–1244, 1998.

[49] B. Maidak, J. Cole, T. Lilburn, C. Parker Jr, P. Saxman, R. Farris, G. Garrity, G. Olsen, T. Schmidt, and J. Tiedje. The rdp-ii (ribosomal database project). *Nucleic Acids Res*, 29(1):173–4, 2001.

[50] S. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[51] E. R. Mardis. Next-generation sequencing platforms. *Annual Review of Analytical Chemistry*, 6:287–303, 2013.

[52] P. O. L. Mark Holder. Phylogeny estimation: traditional and bayesian approaches, 2003.

[53] B. Minh, L. Vinh, A. Haeseler, and H. Schmidt. pIQPNNI: parallel reconstruction of large maximum likelihood phylogenies. *Bioinformatics*, 21(19):3794–3796, 2005.

[54] B. Q. Minh, M. A. T. Nguyen, and A. von Haeseler. Ultrafast approximation for phylogenetic bootstrap. *Molecular biology and evolution*, 2013.

[55] J. N. M.J.L. de Hoon, S. Imoto and S. Miyano. Open source clustering software . *Bioinformatics*, 20(9):1453–1454, 2004.

[56] B. Moret, U. Roshan, and T. Warnow. Sequence-length requirements for phylogenetic methods. *Algorithms in Bioinformatics*, pages 343–356, 2002.

[57] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[58] T. P. Niedringhaus, D. Milanova, M. B. Kerby, M. P. Snyder, and A. E. Barron. Landscape of Next-Generation Sequencing Technologies. *Anal. Chem.*, 83(12):4327–4341, May 2011.

[59] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi; http://www.nvidia.de/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2009.

[60] NVIDIA. Nvidia's next generation cuda compute architecture: Kepler gk110; http://www.nvidia.com/content/pdf/kepler/nvidia-kepler-gk110-architecture-whitepaper.pdf, 2012.

[61] NVIDIA. Cuda c programming guide (design guide, version 5.5); http://docs.nvidia.com/cuda/pdf/cuda_c_programming_guide.pdf, 2013.

[62] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[63] H. Philippe and M. Blanchette. Overview of the first phylogenomics conference. *BMC Evolutionary Biology*, 7(Suppl 1):S1, 2007.

[64] S. Pond and S. Muse. Column sorting: Rapid calculation of the phylogenetic likelihood function. *Systematic biology*, 53(5):685–692, 2004.

[65] F. Pratas, P. Trancoso, L. Sousa, A. Stamatakis, G. Shi, and V. Kindratenko. Fine-grain parallelism using multi-core, cell/be, and gpu systems. *Parallel Computing*, 38(8):365–390, 2012.

[66] M. Price, P. Dehal, and A. Arkin. FastTree 2–Approximately Maximum-Likelihood Trees for Large Alignments. *PLoS ONE*, 5(3):e9490, 2010.

[67] R. A. Pyron and J. J. Wiens. Large-scale phylogenetic analyses reveal the causes of high tropical amphibian diversity. *Proceedings of the Royal Society B: Biological Sciences*, 280(1770), 2013.

[68] C. Quast, E. Pruesse, P. Yilmaz, J. Gerken, T. Schweer, P. Yarza, J. Peplies, and F. O. Glckner. The silva ribosomal rna gene database project: improved data processing and web-based tools. *Nucleic Acids Research*, 41(Database-Issue):590–596, 2013.

[69] D. Robinson and L. Foulds. Comparison of phylogenetic trees. *Math. Biosci*, 53(1-2):131–147, 1981.

[70] S. Roch. A Short Proof that Phylogenetic Tree Reconstruction by Maximum Likelihood Is Hard. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pages 92–94, 2006.

[71] T. Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC Bioinformatics*, 12(1):221, 2011.

[72] F. Ronquist and J. Huelsenbeck. MrBayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics*, 19(12):1572–1574, 2003.

[73] F. Ronquist, M. Teslenko, P. van der Mark, D. L. Ayres, A. Darling, S. Hhna, B. Larget, L. Liu, M. A. Suchard, and J. P. Huelsenbeck. Mrbayes 3.2: Efficient bayesian phylogenetic inference and model choice across a large model space. *Systematic Biology*, 2012.

[74] B. Roure, D. Baurain, and H. Philippe. Impact of missing data on phylogenies inferred from empirical phylogenomic data sets. *Molecular Biology and Evolution*, 30(1):197–214, 2013.

[75] J. Shendure and H. Ji. Next-generation DNA sequencing. *Nature biotechnology*, 26(10):1135–1145, Oct. 2008.

[76] H. Shimodaira. An Approximately Unbiased Test of Phylogenetic Tree Selection. *Systematic Biology*, 51:492–508, 2002.

[77] H. Shimodaira and M. Hasegawa. Multiple comparisons of log-likelihoods with applications to phylogenetic inference. *Molecular Biology and Evolution*, 16(8):1114, 1999.

[78] H. Shimodaira and M. Hasegawa. CONSEL: for assessing the confidence of phylogenetic tree selection. *Bioinformatics*, 17(12):1246–1247, 2001.

[79] M. Simonsen, T. Mailund, and C. N. S. Pedersen. Building very large neighbour-joining trees. In A. L. N. Fred, J. Filipe, and H. Gamboa, editors, *BIOINFORMATICS*, pages 26–32. INSTICC Press, 2010.

[80] S. Smith, J. Beaulieu, A. Stamatakis, and M. Donoghue. Understanding angiosperm diversification using small and large phylogenetic trees. *American Journal of Botany*, pages ajb–1000481v1, 2011.

[81] S. A. Smith, J. M. Beaulieu, and M. J. Donoghue. Mega-phylogeny approach for comparative biology: an alternative to supertree and supermatrix approaches. *BMC Evolutionary Biology*, 9(37), 2009.

[82] S. A. Smith and C. W. Dunn. Phyutility: a phyloinformatics tool for trees, alignments and molecular data. *Bioinformatics*, 24(5):715–716, 2008.

[83] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[84] S. Smits and C. Ouverney. Phylometrics: a pipeline for inferring phylogenetic trees from a sequence relationship network perspective. *BMC Bioinformatics*, 11 Suppl 6, 2010.

[85] A. Stamatakis. Parsimonator; https://github.com/stamatak/parsimonator-1.0.2.

[86] A. Stamatakis. Phylogenetic models of rate heterogeneity: A high performance computing perspective. In *In Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[87] A. Stamatakis. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*, 22(21):2688–2690, 2006.

[88] A. Stamatakis. *Phylogenetic Search Algorithms for Maximum Likelihood*, pages 547–577. John Wiley & Sons, Inc., 2011.

[89] A. Stamatakis. Orchestrating the phylogenetic likelihood function on emerging parallel architectures. *Bioinformatics–High Performance Parallel Computer Architectures, B. Schmidt, Ed. CRC Press*, pages 85–115, 2012.

[90] A. Stamatakis and A. J. Aberer. Novel parallelization schemes for large-scale likelihood-based phylogenetic inference. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1195–1204, 2013.

[91] A. Stamatakis, A. J. Aberer, C. Goll, S. A. Smith, S. A. Berger, and F. Izquierdo-Carrasco. RAxML-Light: a tool for computing terabyte phylogenies. *Bioinformatics*, 28(15):2064–2066, 2012.

[92] A. Stamatakis and N. Alachiotis. Time and memory efficient likelihood-based tree searches on phylogenomic alignments with missing data. *Bioinformatics*, 26(12):132–139, 2010.

[93] A. Stamatakis, F. Blagojevic, D. Nikolopoulos, and C. Antonopoulos. Exploring new search algorithms and hardware for phylogenetics: Raxml meets the ibm cell. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 48(3):271–286, 2007.

[94] A. Stamatakis and F. Izquierdo-Carrasco. Result verification, code verification and computation of support values in phylogenetics. *Brief. Bioinformatics*, 12(3):270–279, May 2011.

[95] A. Stamatakis, T. Ludwig, and H. Meier. RAxML-III: A Fast Program for Maximum Likelihood-based Inference of Large Phylogenetic Trees. *Bioinformatics*, 21(4):456–463, 2005.

[96] A. Stamatakis, T. Ludwig, H. Meier, and M. J. Wolf. AxML: A Fast Program for Sequential and Parallel Phylogenetic Tree Calculations Based on the Maximum Likelihood Method. In *Proceedings of 1st IEEE Computer Society Bioinformatics Conference (CSB2002)*, pages 21–28, 2002.

[97] A. Stamatakis and M. Ott. Efficient computation of the phylogenetic likelihood function on multi-gene alignments and multi-core architectures. *Phil. Trans. R. Soc. series B, Biol. Sci.*, 363:3977–3984, 2008.

[98] A. Stamatakis and M. Ott. Load Balance in the Phylogenetic Likelihood Kernel. In *Proceedings of ICPP 2009*, 2009. accepted for publication.

[99] M. A. Suchard and A. Rambaut. Many-core algorithms for statistical phylogenetics. *Bioinformatics*, 25(11):1370–1376, 2009.

[100] J. Sumner and M. Charleston. Phylogenetic estimation with partial likelihood tensors. *Journal of theoretical biology*, 262(3):413–424, 2010.

[101] S. Sunagawa et al. Metagenomic species profiling using universal phylogenetic marker genes. *Nat Meth*, in press.

[102] C. J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *MICRO*, pages 306–317. ACM/IEEE, 2002.

[103] J. Thompson, T. Gibson, and D. Higgins. Multiple sequence alignment using clustalw and clustalx. *Curr Protoc Bioinformatics*, Chapter 2, 2002.

[104] L. S. Vinh, H. A. Schmidt, and A. von Haeseler. Phynav: A novel approach to reconstruct large phylogenies. In C. Weihs and W. Gaul, editors, *GfKl*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 386–393. Springer, 2004.

[105] J. S. Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.*, 2(4):305–474, January 2008.

[106] T. Wheeler. Large-scale neighbor-joining with ninja. In S. Salzberg and T. Warnow, editors, *Algorithms in Bioinformatics*, volume 5724 of *Lecture Notes in Computer Science*, pages 375–389. Springer Berlin / Heidelberg, 2009.

[107] S. Whelan, P. I. W. de Bakker, and N. Goldman. Pandit: a database of protein and associated nucleotide domains with inferred trees. *Bioinformatics*, 19(12):1556–1563, 2003.

[108] D. Wu, A. Hartman, N. Ward, and J. Eisen. An automated phylogenetic tree-based small subunit rrna taxonomy and alignment pipeline (stap). *PLoS One*, 3(7):e2566, 2008.

[109] J. Xu and R. J. Lipton. On fundamental tradeoffs between delay bounds and computational complexity in packet scheduling algorithms. *IEEE/ACM Trans. Netw.*, 13:15–28, February 2005.

[110] Z. Yang. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites. *J. Mol. Evol.*, 39:306–314, 1994.

[111] Z. Yang. Among-site rate variation and its impact on phylogenetic analyses. *Trends in Ecology & Evolution*, 11(9):367 – 372, 1996.

[112] Z. Yang. *Computational Molecular Evolution.* Oxford University Press, Oxford, 2006.

[113] J. Zhang and A. Stamatakis. The multi-processor scheduling problem in phylogenetics. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 691–698. IEEE, 2012.

[114] Y. Zhang, I. Sinclair, Mark, and A. Chien. Improving performance portability in opencl programs. In J. Kunkel, T. Ludwig, and H. Meuer, editors, *Supercomputing*, volume 7905 of *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin Heidelberg, 2013.

[115] J. Zhou, X. Liu, D. Stones, Q. Xie, and G. Wang. Mrbayes on a graphics processing unit. *Bioinformatics*, 27(9):1255–1261, 2011.

[116] D. Zwickl. *Genetic Algorithm Approaches for the Phylogenetic Analysis of Large Biological Sequence Datasets under the Maximum Likelihood Criterion.* PhD thesis, University of Texas at Austin, April 2006.

[117] D. Zwickl and D. Hillis. Increased taxon sampling greatly reduces phylogenetic error. *Systematic Biology*, 51(4):588–598, 2002.