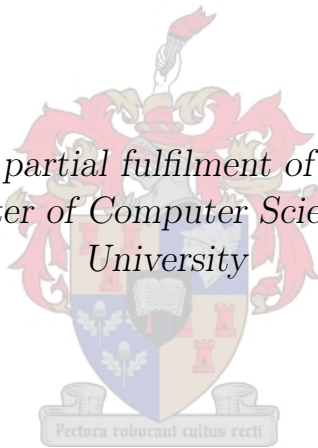


Parallel likelihood calculations for phylogenetic trees

by

Peter Hayward

*Thesis presented in partial fulfilment of the requirements for
the degree of Master of Computer Science at Stellenbosch
University*



Computer Science Division in the Department of Mathematical Sciences,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Prof. K. Scheffler

September 2011

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature:

P.J. Hayward

Date: 21 September 2011
.....

Copyright © 2011 Stellenbosch University
All rights reserved.

Abstract

Parallel likelihood calculations for phylogenetic trees

P.J. Hayward

*Computer Science Division in the Department of Mathematical Sciences,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.*

Thesis: MSc (Computer Science)

September 2011

Phylogenetic analysis is the study of evolutionary relationships among organisms. To this end, phylogenetic trees, or evolutionary trees, are used to depict the evolutionary relationships between organisms as reconstructed from DNA sequence data. The likelihood of a given tree is commonly calculated for many purposes including inferring phylogenies, sampling from the space of likely trees and inferring other parameters governing the evolutionary process. This is done using Felsenstein's algorithm, a widely implemented dynamic programming approach that reduces the computational complexity from exponential to linear in the number of taxa. However, with the advent of efficient modern sequencing techniques the size of data sets are rapidly increasing beyond current computational capability.

Parallel computing has been used successfully to address many similar problems and is currently receiving attention in the realm of phylogenetic analysis. Work has been done using data decomposition, where the likelihood calculation is parallelised over DNA sequence sites. We propose an alternative way of parallelising the likelihood calculation, which we call *segmentation*, where the tree is broken down into subtrees and the likelihood of each subtree is calculated concurrently over multiple processes. We introduce our proposed system, which aims to drastically increase the size of trees that can be practically used in phylogenetic analysis. Then, we evaluate the system on large phylogenies which are constructed from both real and synthetic data, to show that a larger decrease of run times are obtained when the system is used.

Uittreksel

Parallele waarskynlikheidsberekeninge vir filogenetiese bome

(“Parallel likelihood calculations for phylogenetic trees”)

P.J. Hayward

*Afdeling Rekenaarwetenskap in die Departement van Wiskundige Wetenskappe,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.*

Tesis: MSc (Rekenaarwetenskap)

September 2011

Filogenetiese analise is die studie van evolusionêre verwantskappe tussen organismes. Filogenetiese of evolusionêre bome word aangewend om die evolusionêre verwantskappe, soos herwin vanuit DNS-kettings data, tussen organismes uit te beeld. Die aanneemlikheid van ’n gegewe filogenie word oor die algemeen bereken en aangewend vir menigte doeleindes, insluitende die afleiding van filogenetiese bome, om te monster vanuit ’n versameling van sulke moontlike bome en vir die afleiding van ander belangrike parameters in die evolusionêre proses. Dit word vermag met behulp van Felsenstein se algoritme, ’n alombekende benaderingwyse wat gebruik maak van dinamiese programmering om die berekeningskompleksiteit van eksponensieel na lineêr in die aantal taxa, te herlei. Desnieteenstaande, het die koms van moderne, doeltreffender orderingsmetodes groter datastelle tot gevolg wat vinnig besig is om bestaande berekeningsvermoë te oorskry.

Parallele berekeningsmetodes is reeds suksesvol toegepas om vele soortgelyke probleme op te los, met groot belangstelling tans in die sfeer van filogenetiese analise. Werk is al gedoen wat gebruik maak van data dekomposisie, waar die aanneemlikheidsberekening oor die DNS basisse geparallelliseer word. Ons stel ’n alternatiewe metode voor, wat ons *segmentasie* noem, om die aanneemlikheidsberekening te parallelliseer, deur die filogenetiese boom op te breek in sub-bome, en die aanneemlikheid van elke sub-boom gelyklopend te bereken oor verskeie verwerkingseenhede. Ons stel ’n stelsel voor wat dit ten doel het om ’n drastiese toename in die grootte van die bome wat gebruik kan word in filogenetiese analise, teweeg te bring. Dan, word ons voorgestelde stelsel op

groot filogenetiese bome, wat vanaf werklike en sintetiese data gekonstrueer is, evalueer. Dit toon aan dat 'n groter afname in looptyd verkry word wanneer die stelsel in gebruik is.

Acknowledgements

I would like to express my sincere gratitude to the following people:

- my supervisor, Prof. Konrad Scheffler, for his mentorship, support and guidance throughout the writing of this thesis;
- my parents, Dan and Jobeth Hayward, who provided enormous support, enabling me to pursue furthering my education;
- my sister and friends who always listened;
- and finally, my proofreaders, Amy Becht, Stephan Gouws, John Gilmore, Danielle du Plessis and Dr. McElory Hoffmann.

Contents

Declaration	i
Abstract	ii
Uittreksel	iii
Acknowledgements	v
Contents	vi
List of Algorithms	ix
List of Figures	x
List of Tables	xiii
Nomenclature	xiv
1 Introduction	1
1.1 Motivation	1
1.2 Problem statement	4
1.3 Research Questions	5
1.4 Research objectives	5
1.5 Thesis overview	6
2 Preliminaries	7
2.1 Likelihood functions	10
2.2 Phylogenetics	11
2.2.1 Sequence alignments	11
2.2.2 Phylogenetic trees	14
2.2.3 Probabilistic models of evolution	16
2.2.4 Felsenstein’s pruning algorithm	18
2.3 Parallel programming	22
2.3.1 Decomposition and distribution	22
2.3.2 Parallel Performance	24

2.4	Cache and memory	25
2.5	Parallelisation approaches in phylogenetics	27
2.5.1	Parallelisation when inferring phylogenies	28
2.5.2	Parallelisation of the likelihood calculation over sequence alignment sites	29
2.5.3	Likelihood parallelisation on GPUs	29
2.6	Summary	30
3	Parallel Likelihood Algorithms	31
3.1	Complexity analyses of Felsenstein's algorithm	32
3.1.1	Estimating the run time	32
3.1.2	Estimating the memory footprint	33
3.2	Effects of the memory footprint on the likelihood algorithm run times	36
3.2.1	Data decomposition of the likelihood problem	36
3.2.2	Likelihood calculation run times and cache limits	41
3.3	Estimating the likelihood calculation run times	44
3.4	Proposed system overview	45
3.4.1	Finding the optimal threshold	47
3.4.2	Phylogenetic tree segmentation algorithm (PTS)	47
3.4.2.1	Segmenting trees	48
3.4.2.2	Substitution vectors and subtree dependence	50
3.4.2.3	Alternative segmentation methods	53
3.4.3	Process Management	53
3.4.3.1	Process blocking and waiting time	54
3.4.3.2	Priority queues	55
3.4.3.3	Priority counter	58
3.4.4	Run time estimation of the segmented phylogeny	60
3.4.5	Blocked subtree segmentation (BSS)	60
3.5	Summary	62
4	Results	65
4.1	Data sets	65
4.2	Speedup as a function of phylogeny size	68
4.2.1	Speedup as a function of the number of processes	68
4.2.2	Run time estimation as a function of the number of processes	71
4.3	Summary	73
5	Conclusion	80
5.1	Summary of findings	80
5.2	Contributions	81
5.3	Suggestions for future work	82

CONTENTS

viii

List of References

84

List of Algorithms

2.1	Recursive likelihood function in Felsenstein's pruning algorithm	19
3.1	Phylogenetic tree segmentation algorithm	49
3.2	Blocked subtree segmentation	61

List of Figures

1.1	An extract of a sequence alignment for the <i>HA</i> gene of the <i>H1N1</i> influenza virus rendered in <i>ClustalX</i> [1].	2
2.1	The sketch of the tree of life as drawn by Darwin in 1837 from his <i>Notebook B</i>	7
2.2	Phylogeny generated from fully sequenced genomes by the <i>ITOL: Interactive Tree of Life</i> project's online phylogenetic analyses tool, as published on their website [2].	8
2.3	Extract from the ITOL phylogeny	9
2.4	A simple tree showing the change of characters in a string over time.	11
2.5	The data preparation process to obtain ungapped alignments for likelihood calculation.	12
2.6	A simple phylogeny with three leaf nodes for a sequence alignment with six sites.	15
2.7	A simple phylogeny τ , with a subtree τ' depicted in dashed lines.	16
2.8	Phylogeny with three leaf nodes and only one site.	20
2.9	$P(\mathcal{S}^0 \theta) = \sum_x \pi_x L_5^0(x)$	21
2.10	$L_0^0(x) = (\sum_y P(y x, t_4) L_4^0(y)) (\sum_z P(z x, t_1) L_1^0(z))$	22
2.11	$k = 1$ is a leaf node	23
2.12	The von Neumann architecture shows the CPU that is made up by the control unit and arithmetic logic unit, main memory, and IO devices.	25
3.1	How the sequence alignment is segmented when performing data decomposition over sites and over the phylogeny.	37
3.2	Data decomposition over sites and sequence of the H1N1 HA gene data set: memory size as a function of the number of processes using a nucleotide model.	38
3.3	Figure 3.2 repeated for a codon model.	38
3.4	Data decomposition performed individually for the tree, model and alignment structures, over sites and sequence of the H1N1 HA gene data set.	40
3.5	HyPhy runs performed using nucleotide models on the cluster at the Viral Evolution Group at the School of Medicine UCSD.	42

3.6	Figure 3.5 zoomed in: showing the non-linear increase in run time observed for the smaller phylogeny sizes when nucleotide models are used.	42
3.7	HyPhy runs performed using codon models on the cluster at the Viral Evolution Group at the School of Medicine UCSD.	43
3.8	Flowchart showing an overview of our proposed parallel likelihood algorithm.	46
3.9	Illustration of the segmentation of this simple phylogeny into three subtrees when $n' = 2$. The original tree has a size of five, since it has five leaf nodes. The algorithm produces three subtrees, two of which have a size of two while the remaining one has a size of three.	49
3.10	Node 4 of a simple phylogeny is chosen as the segmentation point. .	51
3.11	Two new subtrees are produced and a substitution vector is created.	51
3.12	How PTS is used to segment this simple ladder-like phylogeny into three subtrees when $n' = 2$	52
3.13	A visualisation of how the PTS algorithm segmented the tree and how the tasks (subtrees) were assigned to the process queues. The tasks are colour coded relative to the process priority queues they were assigned to. Red was assigned to q_0 , blue to q_1 , green to q_2 , orange to q_3 and purple to q_4	57
3.14	An illustration of how the tasks are distributed over the five processes, stored in the queue of each process. The tasks are shown as boxes and the idle time as dashed lines.	59
3.15	The priority queues used with the BSS heuristic.	63
4.1	Phylogeny constructed for the H1N1 HA gene sequence alignment from the NCBI influenza database.	66
4.2	Phylogeny constructed for the H1N1 NP gene sequence alignment from the NCBI influenza database.	66
4.3	Phylogeny constructed for the HIV-1 ENV gene sequence alignment from the Los Alamos HIV database.	67
4.4	Phylogeny constructed for the HIV-1 POL gene sequence alignment from the Los Alamos HIV database.	67
4.5	Estimated run time speedup as a function of phylogeny size using nucleotide and codon models and for different numbers (p) of processes. Sequence alignments with 10, 100, 500 and 1000 sequences were used. This was done for 300 (blue), 1200 (green) and 3000 (red) sites each. Only the results of the PTS algorithm are shown. Speedup is calculated using our database of run time estimates for single processor likelihood computations on different phylogeny sizes (see Section 3.4.4).	74

4.6	Estimated run time speedup for the H1N1 HA sequence alignments when using nucleotide and codon models over a number of processes ranging from 1 to 50. The results of the PTS and BSS algorithms are shown in blue and green respectively. Speedup is calculated using our database of run time estimates for single processor likelihood computations on different phylogeny sizes (see Section 3.4.4). The blue lines show the coordinates (x, y) , where $x = y$, which is what the speedup would be if linear speedup was obtained.	75
4.7	Estimated run time speedup for the H1N1 NP sequence alignments when using nucleotide and codon models over a number of processes ranging from 1 to 50. The results of the PTS and BSS algorithms are shown in blue and green respectively. Speedup is calculated using our database of run time estimates for single processor likelihood computations on different phylogeny sizes (see Section 3.4.4). The blue lines show the coordinates (x, y) , where $x = y$, which is what the speedup would be if linear speedup was obtained.	76
4.8	Estimated run time speedup for the HIV-1 ENV sequence alignments when using nucleotide and codon models over a number of processes ranging from 1 to 50. The results of the PTS and BSS algorithms are shown in blue and green respectively. Speedup is calculated using our database of run time estimates for single processor likelihood computations on different phylogeny sizes (see Section 3.4.4). The blue lines show the coordinates (x, y) , where $x = y$, which is what the speedup would be if linear speedup was obtained.	77
4.9	Estimated run time speedup for the HIV-1 POL sequence alignments when using nucleotide and codon models over a number of processes ranging from 1 to 50. The results of the PTS and BSS algorithms are shown in blue and green respectively. Speedup is calculated using our database of run time estimates for single processor likelihood computations on different phylogeny sizes (see Section 3.4.4). The blue lines show the coordinates (x, y) , where $x = y$, which is what the speedup would be if linear speedup was obtained.	78
4.10	Estimated waiting times for the H1N1 NP, H1N1 HA, HIV-1 ENV and HIV-1 POL sequence alignments when nucleotide models are used over a number of processes ranging from 1 to 50.	79
4.11	Estimated waiting times for the H1N1 NP, H1N1 HA, HIV-1 ENV and HIV-1 POL sequence alignments when codon models are used over a number of processes ranging from 1 to 50.	79

List of Tables

2.1	An RNA version of the universal genetic code.	13
3.1	The maximum number of sequences that can fit into L1 and L2 when using nucleotide models for: $m = 300$, $m = 1200$, and $m = 3000$	43
3.2	The maximum number of sequences that can fit into L1 and L2 cache when using codons models for: $m = 300$, $m = 1200$ and $m = 3000$	44
4.1	Performance metrics for the experiments where $N = 4$, $m = 300$ and $p = 1$	69
4.2	Performance metrics for the experiments where $N = 4$, $m = 1200$ and $p = 1$	69
4.3	Performance metrics for the experiments where $N = 4$, $m = 3000$ and $p = 1$	70
4.4	Performance metrics for the experiments where $N = 4$, $m = 300$ and $p = 10$	70
4.5	Performance metrics for the experiments where $N = 4$, $m = 1200$ and $p = 10$	70
4.6	Performance metrics for the experiments where $N = 4$, $m = 3000$ and $p = 10$	71
4.7	The mean percentage increase in run time speedups from PTS to BSS on the chosen sequence alignments.	73

Nomenclature

Notation

i, j	indexes of a set, vector, or other non-matrix structure
$a - e$	range of variables from a to e
$v - z$	range of variables from v to z
$\alpha - \eta$	range of variables: $\alpha, \beta, \gamma, \delta, \epsilon$ and η
\mathcal{S}	sequence alignment, is a set of sequences
s_i	a sequence in a sequence alignment
u	a site in a sequence alignment
τ	tree topology
\mathbf{k}	set of nodes of τ
k_i	node of a tree, $k_i \in \mathbf{k}$, only k is used for simplicity in some cases
h	height of a node in the tree
$\mathbf{\kappa}$	set of subtrees of τ
κ_i	subtree, $\kappa_i \in \mathbf{\kappa}$
Δ	set of concurrent tasks
Δ_i	task, $\Delta_i \in \Delta$
$w(\Delta_i)$	waiting time in a queue for task Δ_i
$\alpha(\Delta_i)$	time at which task Δ_i was started
$\omega(\Delta_i)$	time at which task Δ_i was completed
d	the number of dependencies a subtree has
l	level of a node in a tree
\mathbf{t}	vector of branch lengths for τ
t_i	branch length, $t_i \in \mathbf{t}$
n	number of sequences in an alignment, and the number of leaf nodes in a phylogeny
m	number of sites in an alignment
n'	number of leaf nodes in a subtree, or the desired size of a subtree
N	number of characters in the evolutionary model; $N = 4$ for nucleotide modes, and $N = 61$ for codon models

$\boldsymbol{\pi}$	vector of equilibrium frequencies
π_i	equilibrium frequency
\boldsymbol{r}	rate parameters
Q	rate matrix
$P(t_i)$	transition probability matrix given a branch length t_i
$P(\mathcal{S} \theta)$	likelihood of a sequence alignment given a set of model parameters
D	a given data set
θ	set of parameters
p	the number of parallel processors or processes, depending on context
p_i	a specified process
q_i	the process queue for process p_i
$S(p)$	the speedup obtained when using p parallel processors
$T(p)$	the time the algorithm runs on p parallel processors
$\psi(N, m, n)$	computation complexity of the likelihood calculation on a phylogeny
$\phi(N, m, n)$	total memory footprint of the likelihood calculation on a phylogeny
$\varphi(l, d)$	priority counter used in our propose system; it is a function of l and d

Chapter 1

Introduction

1.1 Motivation

With the ever-increasing advancements in sequencing technology the number of genetic sequences that are sequenced each year is growing at an exponential rate [3]. This has given rise to a massive amount of data stored in databases such as *GenBank*, a public genetic sequence database of the *National Institutes of Health (NIH)* [4].

To transform all this data into useful information and make sense of it all is a challenging task. Some of the key questions we must ask are: how do these sequences relate to one another, and how will we model and analyse potential relationships? For this purpose phylogenetic trees, also called phylogenies, are used to depict and evaluate the evolutionary relationships between organisms. Thus the primary purpose of phylogenies is to reconstruct the tree of life, a profound and bold undertaking [5, 6, 7]. Examples of the information we can infer from phylogenies include identifying common ancestors, measuring evolutionary time, improving the accuracy of biological sequence alignment and performing inferences about the evolutionary process and effects of natural selection. This information is typically used in areas such as disease control, genomics, genetics and drug discovery.

The phylogenies are inferred from DNA sequence data sampled from organisms of interest using methods such as distance matrices, likelihoods and Bayesian inference [8]. There are multiple trees that can be constructed from the sequence data, and therefore a measure is needed to compare the possibilities [8, 6]. It has been shown that the general problem of finding the correct tree is NP-complete [9].

It is popular to evaluate the fit of an inferred phylogenetic tree τ to the data D by calculating the *likelihood* $P(D|\tau)$, this being the probability of the data given the tree. The data set D is in the form of a *sequence alignment* (see Figure 1.1, and Section 2.2.1) [8, 10]. Likelihood values are used for many purposes, including finding the tree that provides the best fit to the

sequence data, sampling from the space of possible trees and inferring other parameters governing the evolutionary process. Computing likelihoods is of central importance in many of the statistical frameworks used in phylogenetics such as maximum likelihood and Bayesian inference [8, 10].

Calculating the likelihood of a phylogeny is usually done by means of *Felsenstein's tree pruning algorithm*, a widely implemented dynamic programming approach that reduces the computational complexity of the likelihood computation from exponential to linear in the number of sequences [11, 8, 10]. However, the typical size of a data set on which this algorithm runs is growing beyond current computational capability. Multiple likelihood calculations are run in many of the typical analyses performed in phylogenetics, which compounds the computational problems. Therefore it is of importance to optimize the algorithm. There has been great success in addressing similar problems by using two core methods of parallel computing: data decomposition, and task decomposition [12]. Today these methods are widely used to solve computationally complex problems in bioinformatics and computational biology [13]. Felsenstein's tree pruning algorithm is amongst the problems on which significant work has been done over the last decade to find scalable parallel solutions.

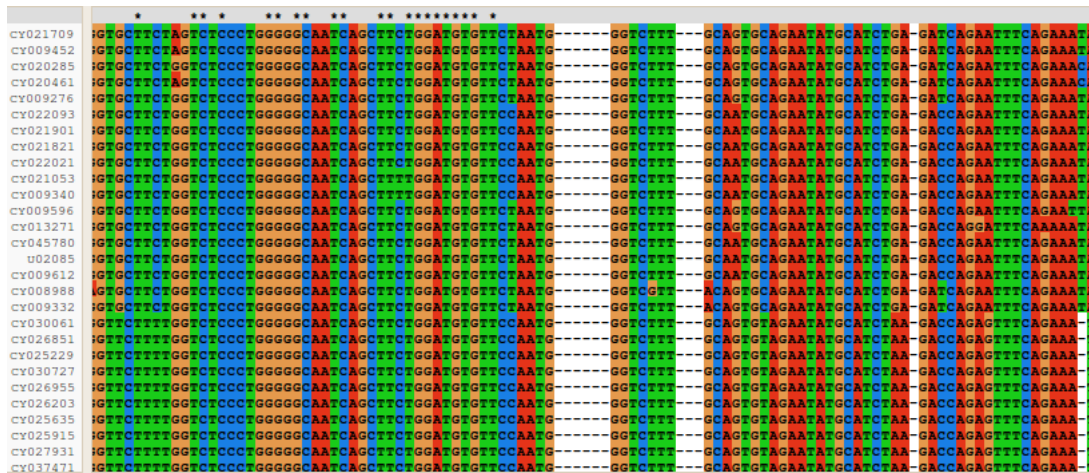


Figure 1.1: An extract of a sequence alignment for the *HA* gene of the *H1N1* influenza virus rendered in *ClustalX* [1].

A sequence alignment consists of multiple DNA sequences, as can be seen in Figure 1.1 (sequences are shown as rows). A DNA sequence is represented as a string of characters, as covered in Section 2.2.1, where each sequence character can be identified by a position index, called a *site* (a column in the alignment). Figure 1.1 shows only a subset of the sequences and sites of this H1N1 HA alignment. The names of the sequences can be seen in the left hand column. Each of the four nucleotides are shown in a separate colour, for example all *C*s are shown in blue. The '*' character indicates sites that

have been fully conserved [14], which means that every character of that site is the same. For example, the first conserved site from the left are all *T*s. The '-' characters in the alignment shown in Figure 1.1 are called *gaps*. Gaps refer to *insertions* and *deletions*, the two types of *substitutions* that occur in sequences [10]. The sequence alignment process calculates the positions of the gaps in the alignment. The sequence alignment process is a way to arrange regions (sections of sites) in the sequences that share similarities in function, structure, or show evolutionary relationships [15]. Gaps are ignored in phylogenetic analysis, since currently there is no good way of incorporating them into the models used for analysis. Therefore any site containing gaps is removed from the alignment once the alignment process has been completed. Figure 1.1 shows only one site that is occupied by gaps and nucleotides. The other sites with gaps will also have at least one sequence with residues at those sites; in this subset they are just not present.

The assumption is made that sites are independent, something which is not strictly biologically true. This independence allows us to calculate the likelihood of each site individually. The first work done on parallel likelihood calculations used this assumption of independence to perform data decomposition over the sites of a sequence alignment. The resulting subsets of sites are distributed to multiple processing cores, after which the likelihood calculation is run concurrently over the subsets [16, 17]. This method is well suited for computer clusters and has been implemented in a number of software packages [18, 17]. We propose an alternative to this method of parallelising over sites.

In our alternative, data decomposition is done over the phylogeny, where the phylogenetic tree is segmented into subtrees and the likelihoods of subtrees are calculated concurrently. The memory footprint of parallelisation over the phylogeny is much smaller than that of parallelisation over sites, as we shall show in Chapter 3. The smaller memory footprint allows for faster computation times, since cache and memory swapping is reduced, enabling the processing of much larger phylogenies.

The upper limit of the number of species is much greater than the upper limit of sequence lengths that are typically used. It is estimated that there are between 10 and 100 million species; to date, 1.7 million are known [7]. Currently, large sequence alignments contain thousands to tens of thousands of sequences, typically of genes. Data sets of human genes or human proteins have an average sequence length of about 3000 sites, but full genomes are very large. The human genome has 2.9 billion base pairs (sites) [19]. These numbers are far too large for current hardware to handle. Today practical likelihood computation times are limited to hundreds of sequences. Our focus, by performing parallelisation over phylogenies, is to improve computation time by segmenting the memory footprint, so that sequence alignment sizes of thousands to tens of thousands can be computed. However, the number of sequences per sequence alignment is only limited to the number of samples taken,

and there is a constant demand to build phylogenies from larger numbers of sequences. By segmenting the memory footprint the algorithm becomes more scalable, and this attribute allows the likelihood calculation to accommodate the ever-increasing sequence alignment size.

1.2 Problem statement

Many phylogenetic analysis problems such as inferring phylogenies, identifying positive selection and testing evolutionary hypotheses can be solved using statistical frameworks, such as maximum likelihood and Bayesian inference. These frameworks are in common use today since they are more accurate, and in many cases more realistic, than other methods. The statistical frameworks require the calculation of the likelihood for a sequence alignment given a phylogeny and a model of evolution. The likelihood calculation is a computationally intensive task, which becomes less tractable as the number of sequences in the given sequence alignment increases. A single likelihood calculation for a sequence alignment of 10 sequences and 300 sites takes roughly 18.1 milliseconds to compute when using a codon model in the *HyPhy* (*Hypothesis Testing using Phylogenies*)¹ package [20]. The computation time is linear in the number of sequences barring effects related to memory. Thus we find that if the number of sequences in the alignment is increased to 100 it takes roughly 178.6 milliseconds to calculate the likelihood. By itself this is not a lot. However, in typical applications the likelihood has to be computed multiple times. For example, HyPhy computed the likelihood calculation 520 times for the sequence alignment of 10 sequences and 300 sites according to the call-graph generator of the software package *Valgrind* [21]. The problem worsens when the phylogeny of interest becomes larger, since the number of likelihood calculations increases as the data set size increases. This makes the processing of large phylogenetic trees less feasible.

Currently, the likelihoods for alignments consisting of *hundreds of sequences* can be computed. This is done on current hardware with state of the art optimisations made to the likelihood computation, including data decomposition over the sites of a sequence alignment. However, the trivial solution of data decomposition over sites is unsatisfactory, since some parts of the likelihood calculation and its memory footprint are duplicated multiple times over the distributed processes.

Our aim is to make the computation of likelihoods on sequence alignments consisting of *thousands to tens of thousands of sequences* feasible. Therefore, our primary concern in this work is studying methods that segment large phylogenies so that the processing of the likelihood calculation and its memory footprint can be distributed over multiple processors more efficiently. Thereby

¹HyPhy is a software package developed for the study of molecular evolution and statistical sequence analysis.

larger phylogenies can be solved, and an improvement of the average run time can be made.

1.3 Research Questions

Our hypothesis is that the likelihood calculation on a large phylogeny can be made tractable by segmenting the phylogeny into subtrees on which separate likelihood calculations can be performed concurrently using multiple processes. We shall call this process parallelisation over the phylogeny.

To date, performing parallelisation over sites has been the way in which reduction of the likelihood calculation run time has been obtained. However, in our alternative the memory footprint of the problem is reduced, and the likelihood computation is distributed even further, as we shall show in Chapter 3. These calculations can also be done concurrently on multiple processors, thereby attaining run time speedup. Therefore, we pose the following set of questions.

- How do we estimate the memory footprint of a likelihood calculation for a given phylogeny?
- How do we estimate run times based on problem size?
- How do the memory footprint reductions of parallelisation over sites and parallelisation over the phylogeny compare?
- How do we obtain the subtree size that will obtain optimal run times?
- How can the phylogeny be segmented once the optimal subtree size is found?
- How will the subtrees be distributed to multiple processes?
- Can the run time of the likelihood calculation on large phylogenies be reduced so that tasks such as inferring large phylogenies can be completed in hours instead of days? For example, the phylogenies shown in Section 4.1 took between 42 hours and 72 hours to infer with the *Phyml* software package [22].

1.4 Research objectives

To answer our research questions we define the following research objectives:

- Study Felsenstein's tree pruning algorithm and the structure of phylogenies, in order to develop a *segmentation algorithm* and a *parallel data distribution strategy* that will segment a phylogenetic tree into smaller

subtrees and then distribute the subtrees over multiple processes. The likelihood calculations done on the subtrees are therefore distributed over the processes.

- Build a simulator in order to *analyse and predict* how the segmentation algorithm and parallel data distribution scheduling strategy performs under specific phylogeny sizes and number of processes for both real and synthetic genetic sequence alignments.

1.5 Thesis overview

This thesis consists of five chapters.

- Chapter 1 – Introduction

The current chapter gives a motivation for our research, pose the problem statement, asks the research questions and finally states the research objectives of this thesis.

- Chapter 2 – Preliminaries

In the preliminaries chapter we cover work required for the understanding of our work, which includes: a basic overview of likelihood functions; discussion on important phylogenetic topics relevant to calculating the likelihood of phylogenies; parallel programming topics related to performance of program run times; an overview of computer cache and memory; and related literature on the parallelisation of phylogenies.

- Chapter 3 – Parallel Likelihood Algorithms

The chapter on parallel likelihood algorithms covers our work on the parallelisation over a phylogeny. We present complexity analyses of Felsenstein's tree pruning algorithm, explain the effects of data decomposition on the likelihood algorithm and propose our solution to the problem.

- Chapter 4 – Results

The results chapter firstly introduces the data sets that were used in the development of our proposed system and the experiments performed to test it, and secondly describes the experiments and discusses the findings.

- Chapter 5 – Conclusion

Finally, the conclusion chapter gives a summary of the findings, and suggests future work.

Chapter 2

Preliminaries

Charles Darwin was one of the first to use a tree structure in biology when he illustrated the idea of *the tree of life*, shown in Figure 2.1, as discussed in his famous work *On the Origin of Species* 140 years ago [5]. The text surrounding

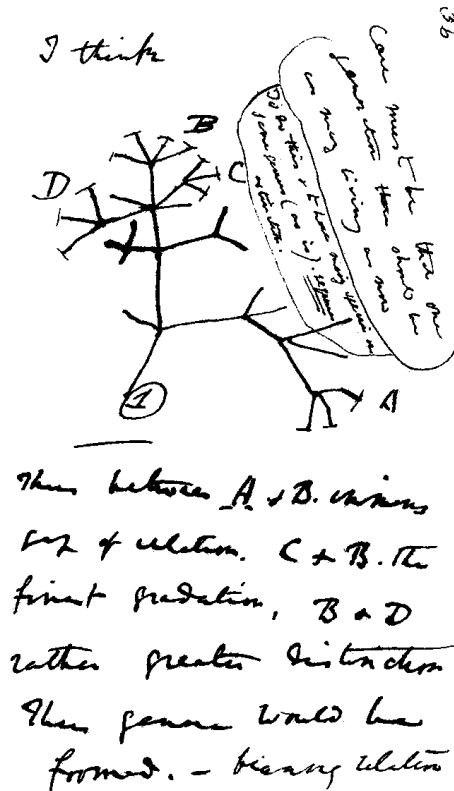


Figure 2.1: The sketch of the tree of life as drawn by Darwin in 1837 from his *Notebook B*.

the tree shows that this was a new hypothesis he was developing and he stated:

Case must be that one generation then should be as many living as now. To do this and to have many species in same genus (as is) requires extinction.

Thus between A + B immense gap of relation. C + B the finest gradation, B + D rather greater distinction. Thus genera would be formed. – bearing relation [page 36 ends - page 37 begins] to ancient types with several extinct forms...

Since then, trees have been a cornerstone of phylogenetics¹; a branch of the life sciences which deals with the study of evolutionary relationships among various groups of organisms [23]. In this biological context, the trees are called *phylogenetic trees* or *phylogenies*. These phylogenies are not known beforehand, and it is the business of phylogenetic analyses to infer phylogenies from sequence data that are sampled in nature. The ultimate goal of phylogenetics is to reconstruct the entire tree of life Darwin suggested [10, 8]. At the time of

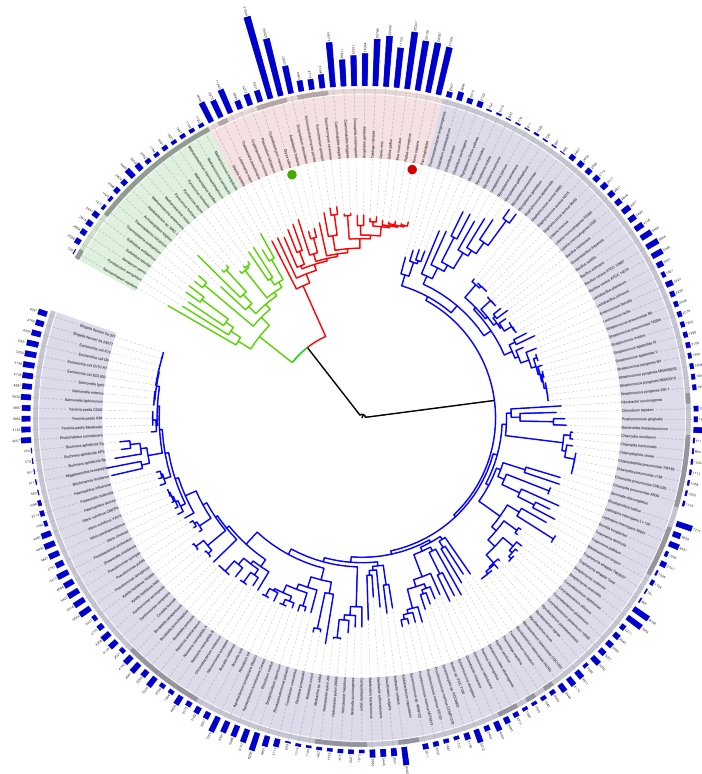


Figure 2.2: Phylogeny generated from fully sequenced genomes by the *ITOL: Interactive Tree of Life* project's online phylogenetic analyses tool, as published on their website [2].

¹From the Greek, *phyle/phylon* meaning "tribe or race", and *genetikos* meaning "relative to birth".

writing, significant work has already been done in this quest. An example of this work is shown by the phylogeny in Figure 2.2, which is inferred from the genomes of organisms that has been fully sequenced by February 2008. The blue bars show the genome size of the organism [2]. Blue branches represent bacteria, red branches represent eukaryotes (including animals and plants) and green branches represent *archaea*. Just for illustration, we focus on the eukaryotes section to show the species names in Figure 2.3. We find *Homo sapiens* (Humans) at the red dot and *Oryza sativa* (Asian rice) at the green dot. Uses

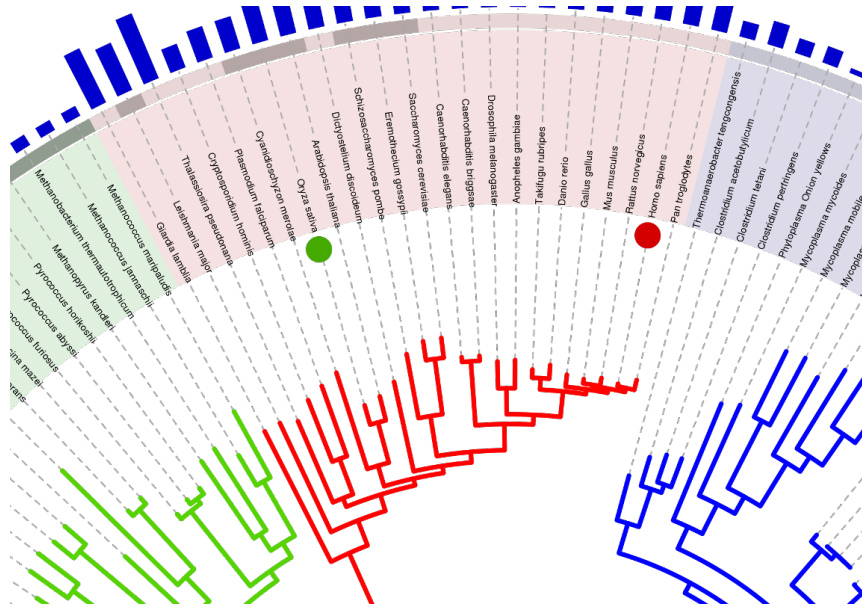


Figure 2.3: Extract from the ITOL phylogeny

of phylogenies, such as the one described above, include using them to aid in more accurate sequence alignment [10] and identifying positive selection.

This thesis is concerned with optimising the calculation of likelihoods for phylogenies by making use of parallel programming. Therefore, in this chapter, we look at some of the key concepts necessary to understand the likelihood calculation and our work in developing a parallel version of the algorithm. Firstly, we give a short introduction to the statistical concept of likelihood in Section 2.1. Thereafter, in Section 2.2, we follow the key concepts used in phylogenetics to compute likelihoods of a phylogeny. Section 2.3 looks at basic parallel programming theory used in our argument. Section 2.4 covers the basics of processor cache and memory, which profoundly influences the performance of algorithms, including the likelihood algorithm. Lastly, Section 2.5 covers related work done on the parallelisation of phylogenetic inference.

2.1 Likelihood functions

In everyday language the term likelihood is often used as a synonym for probability. However, in statistics, likelihood is used to denote the probability of observed values D , given a set of model parameter values θ .

There are many cases where we have a data set of observations D taken from some unknown or partially known system. A model, describing the system, can be constructed consisting of a set of model parameter values θ . Different model parameter values represent the different hypotheses we hold about the system of interest. Given a model, we can calculate how well a model fits the observations by using the *likelihood function* $P(D|\theta)$. We shall describe the process by giving two examples.

Let us consider a data set $D = \{A\}$, meaning that we have observed one character, A . A model must be defined in order to calculate the likelihood of observing A . Therefore, model parameters must be defined. Consider the parameters

$$\boldsymbol{\pi} = [\pi_A \quad \pi_C \quad \pi_G \quad \pi_T] \quad (2.1.1)$$

being the probabilities of observing a character in the alphabet $\{A, C, G, T\}$. Now, let $\pi_A = 0.3$, $\pi_C = 0.2$, $\pi_G = 0.4$, and $\pi_T = 0.1$. Then, the likelihood of observing A is calculated by

$$P(D|\boldsymbol{\pi}) = \pi_A = 0.3. \quad (2.1.2)$$

Thus, the likelihood of observing a character $D = \{x\}$ is

$$P(D|\boldsymbol{\pi}) = \pi_x. \quad (2.1.3)$$

Now, for a slightly more complicated example; say we have two strings of characters observed at two instances of time,

$$D = \begin{Bmatrix} AG \\ CG \end{Bmatrix}, \quad (2.1.4)$$

where rows correspond to sequences and columns to sites. This means that at the first site we see a change from A to C, while at the second site no change is seen. Different sites in a sequence alignment are modelled as being independent.

Now, say we have a tree with only two nodes and one branch connecting them, each having one of the observed strings associated with it, as seen in Figure 2.4. This figure represents the characters at one of the nodes changing into the characters of the other node over some set amount of time (i.e., characters can be substituted over a set amount of time). To model this change from one node to another we need to define model parameters. We shall again use Equation (2.1.1) and the probabilities we assigned to each element of $\boldsymbol{\pi}$

AG ————— CG

Figure 2.4: A simple tree showing the change of characters in a string over time.

as the probability of observing each character. A substitution matrix is used to model the probabilities that a character x will change to character y ; as with the parameters of $\boldsymbol{\pi}$ above, we illustrate the calculation using a set of arbitrarily chosen parameter values:

$$P(y|x) = \begin{matrix} & A & C & G & T \\ \begin{matrix} A \\ C \\ G \\ T \end{matrix} & \begin{pmatrix} 0.5 & 0.166\dot{6} & 0.166\dot{6} & 0.166\dot{6} \\ 0.166\dot{6} & 0.5 & 0.166\dot{6} & 0.166\dot{6} \\ 0.166\dot{6} & 0.166\dot{6} & 0.5 & 0.166\dot{6} \\ 0.166\dot{6} & 0.166\dot{6} & 0.166\dot{6} & 0.5 \end{pmatrix} \end{matrix} \quad (2.1.5)$$

In this matrix, the rows show what the probabilities are of a given character changing to each of the possible characters. For example, the probability that A will not change is read from the first entry $P(A|A) = 0.5$, the probability that an A will change into any of the other characters is read from the next three entries $P(A|C) = P(A|G) = P(A|T) = 0.166\dot{6}$.

Then, the likelihood can be calculated by

$$\begin{aligned} P(D|\boldsymbol{\pi}) &= (\pi_A P(C|A))(\pi_G P(G|G)) \\ &= (0.3 \cdot 0.166\dot{6})(0.4 \cdot 0.5) \\ &= 0.01 \end{aligned} \quad (2.1.6)$$

In practice, parameters such as those in Equations (2.1.1) and (2.1.5) are learned from the data. The process of learning these values involves performing many likelihood calculations for specific parameter values.

2.2 Phylogenetics

This section covers key concepts used in phylogenetics to compute likelihoods of a phylogeny including our data sets (sequence alignment) in Section 2.2.1, phylogenetic trees in Section 2.2.2, probabilistic models of evolution in Section 2.2.3 and Felsenstein's likelihood algorithm in Section 2.2.4.

2.2.1 Sequence alignments

Genetic samples are taken from organisms of interest to us and are put through a process called *sequencing*, which results in genetic sequences we can analyse [24]. Scientists have been performing sequencing since the 1970s. Early

methods based on two-dimensional chromatography were labour intensive and slow, but technological advances in dye-based sequencing and automated analysis systems have resulted in the explosion of sequencing data available [25].

In bioinformatics, it is common to take a data set of such sequences and align them to obtain what is called a *sequence alignment* [10]. An example of an alignment was shown in Figure 1.1. Sequence alignment is a way to identify homologous regions in the sequences (i.e., regions with shared evolutionary ancestry). Such regions can be expected to have similarities in structure and function.

When comparing two sequences, we look for evidence that they have diverged from a common ancestor by a process of mutation and selection [10]. The basic mutational processes are called *insertions* and *deletions*, otherwise known as *indels* [26, 8]. Insertions are when characters are added to a DNA sequence, whereas deletions are when characters are removed from a DNA sequence [26]. They are collectively referred to as *gaps*. The alignment process identifies gaps in the sequences. In both Figures 1.1 and 2.5, gaps are represented with a '-' character.

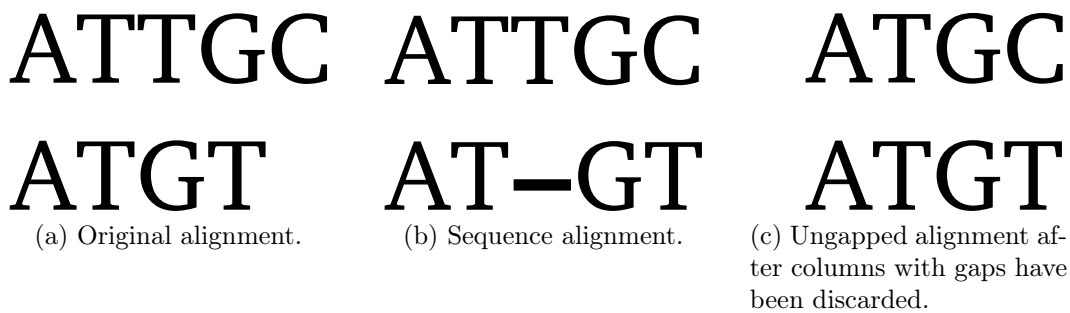


Figure 2.5: The data preparation process to obtain ungapped alignments for likelihood calculation.

The sequence alignment will be denoted as $\mathcal{S} = \{s_0, s_1, \dots, s_n\}$, where s_i is a sequence in the alignment of n sequences. Initially, sequences in the data set may have different lengths. In Figure 2.5a, the first and second sequences have lengths of 5 and 4 characters respectively. When alignment is performed, gaps are introduced into the sequences, as shown in Figure 2.5b. There are situations where we do not know how to handle the gaps in the alignment. In such situations, all columns in the alignment that contain any gaps are discarded, seen in Figure 2.5c. Once aligned, every $s_i \in \mathcal{S}$ has the same length of m residues. The index of a residue is referred to as a *site*.

The genetic sequences that make up the sequence alignment are strings of macromolecules; they can be either *deoxyribonucleic acids (DNA)*, *ribonucleic acids (RNA)*, or *proteins*. In general, we speak of DNA and RNA as *nucleic acids*, where each individual character in a string is called a *nucleotide*.

		2nd base			
		U	C	A	G
1st base	U	UUU (Phe/F)	UCU (Ser/S)	UAU (Tyr/Y)	UGU (Cys/C)
		UUC (Phe/F)	UCC (Ser/S)	UAC (Tyr/Y)	UGC (Cys/C)
		UUA (Leu/L)	UCA (Ser/S)	UAA (Stop)	UGA (Stop)
		UUG (Leu/L)	UCG (Ser/S)	UAG (Stop)	UGG (Trp/W)
	C	CUU (Leu/L)	CCU (Pro/P)	CAU (His/H)	CGU (Arg/R)
		CUC (Leu/L)	CCC (Pro/P)	CAC (His/H)	CGC (Arg/R)
		CUA (Leu/L)	CCA (Pro/P)	CAA (Gln/Q)	CGA (Arg/R)
		CUG (Leu/L)	CCG (Pro/P)	CAG (Gln/Q)	CGG (Arg/R)
	A	AUU (Ile/I)	ACU (Thr/T)	AAU (Asn/N)	AGU (Ser/S)
		AUC (Ile/I)	ACC (Thr/T)	AAC (Asn/N)	AGC (Ser/S)
		AUA (Ile/I)	ACA (Thr/T)	AAA (Lys/K)	AGA (Arg/R)
		AUG[A] (Met/M)	ACG (Thr/T)	AAG (Lys/K)	AGG (Arg/R)
	G	GUU (Val/V)	GCU (Ala/A)	GAU (Asp/D)	GGU (Gly/G)
		GUC (Val/V)	GCC (Ala/A)	GAC (Asp/D)	GGC (Gly/G)
		GUA (Val/V)	GCA (Ala/A)	GAA (Glu/E)	GGA (Gly/G)
		GUG (Val/V)	GCG (Ala/A)	GAG (Glu/E)	GGG (Gly/G)

Table 2.1: An RNA version of the universal genetic code.

Segments of DNA and RNA encode genetic material that is translated into proteins by a many-to-one map called the *genetic code*. These encoded segments are called protein-coding genes. All genes used in the encoding of proteins are made up of a sequence of *codons*. Each codon in turn is made up of a sequence of three nucleotides, and encodes an amino acid. Table 2.1 shows the so-called universal genetic code, where each of the 64 codons designates either one of the 20 standard amino acids or a “stop” signal to terminate translation. Examples of the three letter codon strings can be seen in the table. *UUU*, *UUC*, *UUA* and *UUG* (top left) are examples of this. An abbreviation and character code of the codon type is given in parentheses for each three letter string. For example, the first codon, *UUC*, is a Phenylalanine codon. Thus, the abbreviation in this case is *Phe*, and the character code is *F*. The term *universal code* is used since this code happens to be used by eukaryotes. However, there are many other possible codes [27], thus, this code is not universal in reality.

From a computer scientist’s perspective, the characters in DNA, RNA, and protein strings can be viewed as elements of alphabets. The sizes of these alphabets are an important parameter in the phylogenetic models that we shall encounter throughout this thesis. The two types of phylogenetic models used in phylogenetics to model the evolutionary process are nucleotide models and codon models (amino acids). The alphabet sizes are as follows:

- DNA strings: made up of four characters in the alphabet {A, G, T, C}.
- RNA strings: made up of four characters in the alphabet {U, G, T, C}.
- Amino acids: there are 20 standard².
- Codons: there are 64 codons of which 3 are so-called *stop codons* in the universal genetic code. Other genetic codes do not necessarily have 3

²There are also many non-standard amino acids [28].

stop codons. Usually, stop codons are ignored in codon models in which case the codon alphabet is treated as having 61 characters.

It will be shown in Chapters 3 and 4 that the sizes of the respective alphabets have a big impact on the amount of computation which is performed when computing the likelihoods of a phylogenies.

2.2.2 Phylogenetic trees

To perform phylogenetic analysis on a sequence alignment, one first needs to infer a phylogenetic tree, also known as phylogeny. The inference can be done with various methods including parsimony, distance matrix methods, quartets, maximum likelihood, Hadamard methods, and Bayesian inference [8].

Phylogenetic trees show the relationships between genetic sequences by depicting common ancestors of the sequences and branching events, which happen over evolutionary time. At the highest level of abstraction, a phylogeny consists of a *tree topology* τ and a sequence alignment \mathcal{S} . τ has a discrete structure comprised of a set of *nodes* k and a set of branches. Each branch has an associated *branch length* denoting either the genetic distance or the amount of time separating parent and child nodes. The branch lengths are stored in a set t . A phylogeny can either be a rooted tree or an unrooted tree. If rooted, the phylogeny is a directed tree showing ancestry and has the root as the common ancestor. If unrooted, the phylogeny is an undirected tree where ancestry cannot be deduced, since direction enables us to say which of two nodes comes first. Unrooted trees can be converted to rooted trees by picking one of the nodes as the root. Bifurcating and multifurcating trees are two terms that are frequently encountered describing tree topologies. A phylogeny is called bifurcating if any node in the tree can have at most two child nodes, otherwise it is called multifurcating. A multifurcating tree is typically constructed when the data does not provide enough information to construct a bifurcating tree [29]. The differences between the two has little impact on the calculation of a likelihood, apart from a slight change that needs to be made to Felsenstein's pruning algorithm when working with multifurcating trees (see Section 2.2.4). Figure 2.6 shows a possible rooted bifurcating tree for a small sequence alignment. *Nodes*, denoted k , in a phylogeny can be grouped into two categories: *inner nodes*, and *leaf nodes*. Leaf nodes (taxonomic units) represent the samples taken from organisms, and inner nodes (hidden taxonomic units) represent their ancestors. Each leaf node is, therefore, paired up with one of the sequences in the sequence alignment (this is a one-to-one relationship). A taxon (plural taxa) is a taxonomic unit for a population, or group, of organisms that are usually inferred to be phylogenetically related [30]. A node in the tree topology represents a taxon. The sequences of the inner nodes are not known, but inferred, as the name hidden taxonomic unit implies. Given any sequence alignment of n sequences, τ will have n leaf nodes and $n - 1$ inner nodes, $2n - 1$

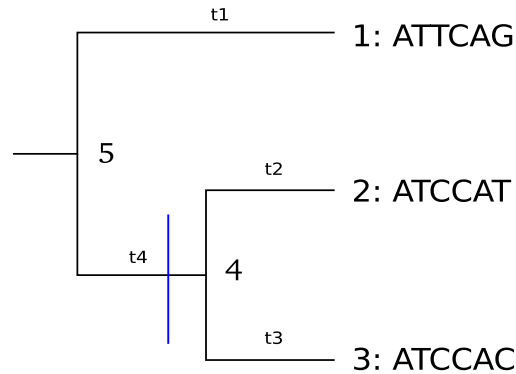


Figure 2.6: A simple phylogeny with three leaf nodes for a sequence alignment with six sites.

nodes in total, if τ is a rooted bifurcating tree. If τ is an unrooted bifurcating tree, the root is absent; therefore, it will have $2n - 2$ nodes, with n leaves and $n - 2$ inner nodes. A standard concept in graph theory is that a node is situated at a specific *depth* in the tree. The set of all nodes at that depth is referred to as a *level*. The root node of the tree is at depth 0, each branch taken down the tree adds one level of depth. For example, in Figure 2.6 node 5 is the root node found at depth 0, nodes 1 and 4 are each one branch down and thus at depth 1; finally, nodes 2 and 3 are down one level further, thus, both are at depth 2.

Two or more nodes will be directly connected via branches from a parent node, which is considered their most recent common ancestor in a rooted tree. For example, in Figure 2.6, node 2 and node 3 are directly connected to their most recent common ancestor, node 4. Branch lengths show the amount of *evolutionary distance*, which is the *rate of change* (covered in Section 2.2.3) multiplied by *time*. If the rate is constant over the whole phylogeny, we can think of evolutionary distance as *evolutionary time*. For example, in Figure 2.6, t_1 is the amount of evolutionary distance between node 4 and node 3.

Any phylogeny can be broken down into smaller trees called *subtrees*. A subtree τ' , of a tree τ , consists of any node k in τ and all the descendent nodes of node k [31]. For example, in Figure 2.7, the root node of the subtree τ' (depicted with dashed lines) is node 4. So, the nodes of τ' , in this case, is 2, 3, and 4. Subtrees allow us to perform the tree segmentation, Section 3.4, which is what we propose in this thesis. Felsenstein's pruning algorithm is used to compute the likelihood of a phylogeny from sequence data. We shall discuss Felsenstein's algorithm in Section 2.2.4.

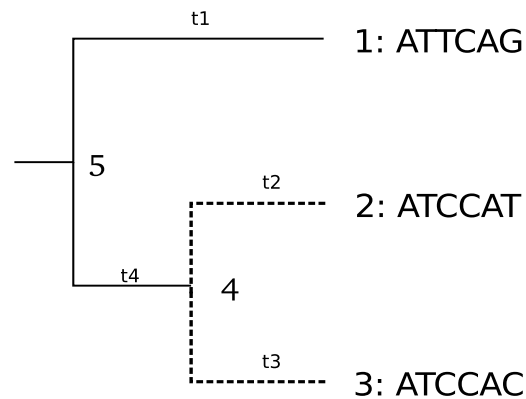


Figure 2.7: A simple phylogeny τ , with a subtree τ' depicted in dashed lines.

2.2.3 Probabilistic models of evolution

Section 2.1 illustrated a simple example in which we calculated the likelihood of a sequence alignment given a model. We showed that the likelihood of how well the model fits the data can be calculated with the likelihood function $P(D|\theta)$. Now, we discuss the standard approach of modeling evolution as a continuous-time Markov process. This leads to ways of parameterising the expression for $P(y|x)$, instead of picking arbitrary values as in Equation (2.1.5).

To date, many probabilistic models of evolution have been suggested to describe the evolutionary process. Each of these attempts to describe some known feature(s) of the evolutionary process. To convey the important properties of these probabilistic evolutionary models, we shall describe the most general of the models, the *general time-reversible (GTR) model*. There are two different GTR models, one for nucleotides and one for codons. The GTR models are the most realistic of all the suggested models, since all the model parameters are inferred from the alignment, and no artificial constraints are imposed as with the simpler, more specialised models.

The GTR model consists of the following components, irrespective of the alphabet size:

- The *tree topology* τ and its associated *vector of branch lengths* \mathbf{t} , as described in Section 2.2.2.
- The *rate matrix* Q models the evolutionary process as a Markov process that describes evolution along τ and the rate of change between characters along the alignment [32, 33, 10, 8]. When describing a nucleotide model, the GTR model defines Q as a 4×4 matrix having 12 independent

parameters, as shown below in Equation (2.2.1).

$$Q = \begin{bmatrix} \cdot & \alpha\pi_G & \beta\pi_C & \gamma\pi_T \\ \alpha\pi_A & \cdot & \delta\pi_C & \epsilon\pi_T \\ \beta\pi_A & \delta\pi_G & \cdot & \eta\pi_T \\ \gamma\pi_A & \epsilon\pi_G & \eta\pi_C & \cdot \end{bmatrix} \quad (2.2.1)$$

If Q were describing a codon model then it would have been a 61×61 matrix, since the codon alphabet contains 61 characters, as we covered in Section 2.2.1. Individual elements in Q represent the instantaneous rates of change from one character in the target row to another character in the target column. The rows are ordered from top to bottom and the columns from left to right, where the character order is A, G, C, and T in both cases. For example, $\alpha\pi_G$ is the substitution rate between A and G. Each element is the product of one of six independent rate parameters $\mathbf{r} = [\alpha \ \beta \ \gamma \ \delta \ \epsilon \ \eta]$, and an *equilibrium frequency*, $\boldsymbol{\pi} = [\pi_A \ \pi_G \ \pi_C \ \pi_T]$. Both \mathbf{r} and $\boldsymbol{\pi}$ are inferred from the sequence alignment. Usually one of the parameters in \mathbf{r} is fixed to 1 due to the fact that Q and t are confounded in Equation (2.2.2). The elements of $\boldsymbol{\pi}$ are the prior probabilities of observing each character. The diagonal elements of Q are treated as special cases, since, in these cases, there are no character changes. The \cdot is a placeholder for the additive inverse of the sum of the other elements in the row. In the case of the first row, it would be $-(\alpha\pi_G + \beta\pi_C + \gamma\pi_T)$. Therefore, the sum of the elements of a row is 0. This is needed so that the transition probability matrix (discussed hereafter) has elements that are valid probabilities.

Given a model $\theta = \{\tau, \mathbf{t}, Q\}$, the probability of changing from character x to character y over a specified period of evolutionary time t can be calculated by the *transition probability matrix*. The transition probability matrix, which is a function of evolutionary time, is calculated by matrix exponentiation [8]:

$$P(t_i) = e^{Qt_i}. \quad (2.2.2)$$

This is done numerically by finding the eigenvalues λ_j and eigenvectors of the rate matrix Q ,

$$Q = UD(\lambda_j)U^{-1}, \quad (2.2.3)$$

where $D(\lambda_j)$ is a diagonal matrix of the eigenvalues of Q , and U is a matrix of which the columns are eigenvectors of Q . $P(t_i)$ is, then, obtained by replacing the elements of $D(\lambda_j)$ in Equation (2.2.3) with $\exp(\lambda_j t_i)$. It will later become important to know that the asymptotic complexity of this rate matrix exponentiation is $O(N^3)$, where N is the alphabet size. Now, the transition probability $P(y|x, t_i)$ is the probability that the sequence y evolved from the ancestor sequence x along a branch of length t_i , which can be read from $P(t_i)$.

2.2.4 Felsenstein's pruning algorithm

In 1981 *Joseph Felsenstein* published a *dynamic programming* algorithm that computes the likelihood of a sequence alignment given a phylogeny and a probabilistic model of evolution. This algorithm, named the *pruning* algorithm, was based on the so-called *peeling* algorithm used for computing likelihoods on pedigrees in human genetics [8]. The pruning algorithm reduced the computational complexity of the likelihood computation from exponential to linear in the number of sequences (i.e., $O(n)$ where n is the number of sequences). The computation time is discussed further in Section 3.1.1. It has become so widely used and his work on the algorithm has been so widely cited that it has become known as *Felsenstein's pruning algorithm* [11, 8]. We shall first define Felsenstein's algorithm, then show an example of how the algorithm is performed on a small data set, and, lastly, explain why it is a dynamic programming algorithm.

The likelihood of the given sequence alignment, which is a set of sequences $\mathcal{S} = \{s_0, s_1, \dots, s_{n-1}\}$, given the model θ , is

$$P(\mathcal{S}|\theta) = \prod_{u=0}^{m-1} P(\mathcal{S}^u|\theta) \quad (2.2.4)$$

where m is the number of sites, and u represents a unique site number in the sequence alignment. For a given site u the likelihood of a site is calculated by

$$P(\mathcal{S}^u|\theta) = \sum_x \pi_x L_k^u(x) \quad (2.2.5)$$

where the subscript of L identifies a node of the tree; the computation starts at the root node, which is labelled k . The sum is taken over all possible character values x at a node. Characters can be nucleotides or codons. π_x is the equilibrium frequency of the character x in the model. In general, L_k^u is calculated by

$$L_k^u(x) = \begin{cases} 1 & \text{if } k \text{ is a leaf node and } x = s_j^u \\ 0 & \text{if } k \text{ is a leaf node and } x \neq s_j^u \\ \prod_i (\sum_y P(y|x, t_i) L_i^u(y)) & \text{if } k \text{ is not a leaf node} \end{cases} \quad (2.2.6)$$

for any given k , where the product is over all immediate descendants (child nodes) i of node k . This computes the probability of observing x at a specified node k at site u in the alignment, conditional on the observations at the leaves that are descendants of k . $L_k^u(x)$ is a recursive function that traverses over the phylogeny in *post-order*. The termination condition is met when a leaf node is reached, at which point the sequence s_j assigned to that leaf is checked at site u . j is the index of the assigned sequence in the alignment. If the character found at s_j^u is equal to x , then a probability of 1 is returned, otherwise, a probability of 0 is returned. If k is not a leaf node, the product of the events

that have taken place at all the descendants of k , given that x was observed, is calculated. The lengths of the branches leading from k to its immediate descendants i are denoted by t_i . To be able to calculate the likelihood of both a bifurcating and multifurcating tree (i.e., the general case), the product over all i must be computed. Usually, the sum-product is written for a bifurcating tree. This convention is adopted in the illustrative example of Figure 2.10. At each i , the sum-product of $P(y|x, t_i)$ and $L_i^u(y)$ is taken over every character y . The pseudo-code for $L_k^u(x)$ can be written as shown below in Algorithm (2.1)

Algorithm 2.1 Recursive likelihood function in Felsenstein's pruning algorithm

```

1: function L( $k, x$ )
2:   if  $k$  is a leaf node then
3:     if  $k = x$  then
4:       return 1
5:     else
6:       return 0
7:     end if
8:   else
9:     return  $\prod_i (\sum_y P(y|x, t_i) L(i, y))$ 
10:  end if
11: end function

```

For clarity, Figures 2.8-2.11 gives an illustrated example of how Felsenstein's algorithm calculates the likelihood for a single site. Starting with Figure 2.8, suppose we have a tree topology with three leaf nodes, a DNA sequence alignment \mathcal{S} with only one site, and we are using a nucleotide model. Q will be a 4×4 rate matrix, since a nucleotide model is used. Our model θ consists of the tree topology τ , branch lengths $\mathbf{t} = [t_1 \ t_2 \ t_3 \ t_4]$, the rate matrix Q , and a vector of equilibrium frequencies $\boldsymbol{\pi} = [\pi_A \ \pi_G \ \pi_C \ \pi_T]$, as covered in Section 2.2.3. For convenience, we also show the names of the nodes, $k = 1$ to $k = 5$, in this first figure. We omit them in the figures that follow to avoid clutter. To calculate the likelihood, Equation (2.2.4) is used. Equation (2.2.5) will only be run once, since \mathcal{S} has only one site, so there will be no product of site likelihoods in this example. Figure 2.9 shows Equation (2.2.5) starting at the root node $k = 5$ of site 0. Since this is a nucleotide model, x has four possible values, and therefore, the sum-product of Equation (2.2.5) is done for all four possibilities. Thus, $L_k^u(x)$ that is implemented as Algorithm (2.1) is called four times. The root node has two children: $k = 1$, and $k = 4$. Figure 2.10 shows how $L_k^u(x)$ is called for any given x , and that, itself, is a product of the sum-products of the child nodes of node k . The transition probability of each possible character is calculated at any given child node, given the character

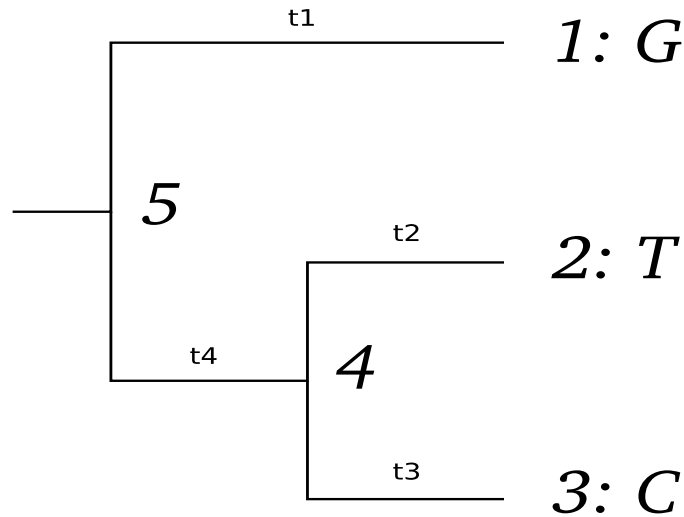


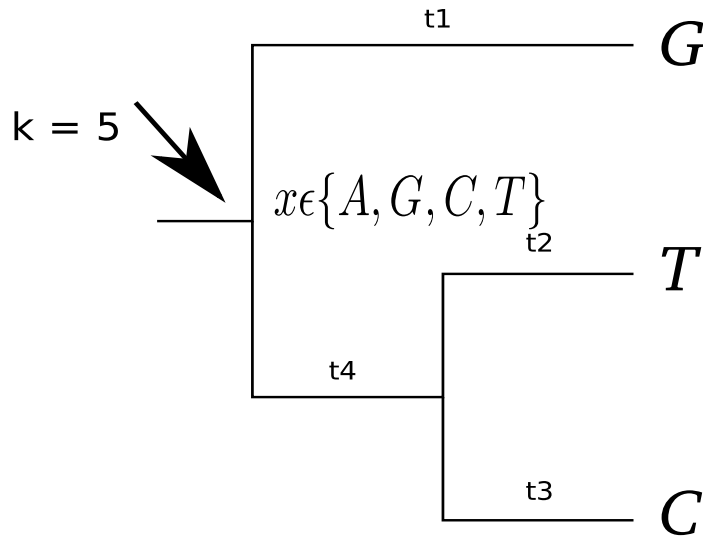
Figure 2.8: Phylogeny with three leaf nodes and only one site.

seen at the parent node and the branch length, for example, $P(y|x, t_4)$. L_k^u is called yet again for each possible character at the child node, making the algorithm recursive. Now, take the node $k = 1$ as an example. Figure 2.11 shows that L_k^u has reached the termination condition, since 1 is a leaf node. This point will be reached four times for $k = 1$. One of the four times z will be equal to G and a probability of 1 will be returned as explained above; the remaining three times z is equal to one of the other nucleotides and a probability of 0 is returned, giving:

$$L_1^0(z) = \begin{cases} 1 & \text{if 1 is a leaf node and } z = G \\ 0 & \text{if 1 is a leaf node and } z \neq G \end{cases}$$

Now that we have shown how Felsenstein's algorithm works, it is easier to understand why the algorithm is categorised as a dynamic programming algorithm. The class of algorithms called *bottom-up dynamic programming algorithms* can be applied to any recursive algorithm³, if the algorithm can, at any point, afford to record computed values at previous points [31]. Such an algorithm starts with the smallest sub-problems and uses the solutions of the sub-problems to solve larger sub-problems. The solution to the whole problem has been obtained once every sub-problem has been solved. For some problems, only a subset of the problems need to be recorded, such as problems

³All tree traversal algorithms are recursive, but can be rewritten as iterative algorithms if needed [31].

Figure 2.9: $P(\mathcal{S}^0|\theta) = \sum_x \pi_x L_5^0(x)$

where only the most recent sub-problems are needed to reach a solution. Implementations of these dynamic programming algorithms become even more efficient when only the necessary sub-problem solutions are recorded, since the implementation uses less memory.

Felsenstein's algorithm is a bottom-up dynamic programming algorithm, because it relies on dividing the problem into sub-problems in order to obtain a solution more efficiently. The sub-problems are obtained by using post-order traversal to traverse the phylogeny from the leaves to the root. The smallest sub-problem is to calculate the likelihood at a leaf node. In the example above, this is done by $L_1^0(z)$, $L_2^0(z)$, and $L_3^0(z)$. The second smallest sub-problem is to calculate the likelihood of the subtree formed by the ancestor of the leaf node and its descendants. For example, $L_4^0(z)$, since node 4 is the ancestor of nodes 2 and 3. The third smallest sub-problem would be to calculate the likelihood of the subtree formed by the ancestor of the leaf node's ancestor, and so forth. So, the likelihoods are known for each subtree where the root node is a descendant of k when node k is visited. Therefore, the likelihood calculation does not need to be recomputed at each descendant of k . Avoiding the need to recompute the likelihoods of all descendant subtrees from each node in the phylogeny is how Felsenstein's algorithm reduces the problem of computing the likelihood of a phylogeny from exponential to linear in number of sequences. How bottom-up dynamic programming works and the fact that Felsenstein's algorithm is such an algorithm is important, since this is the same approach we use in our *tree segmentation algorithm*, which is described

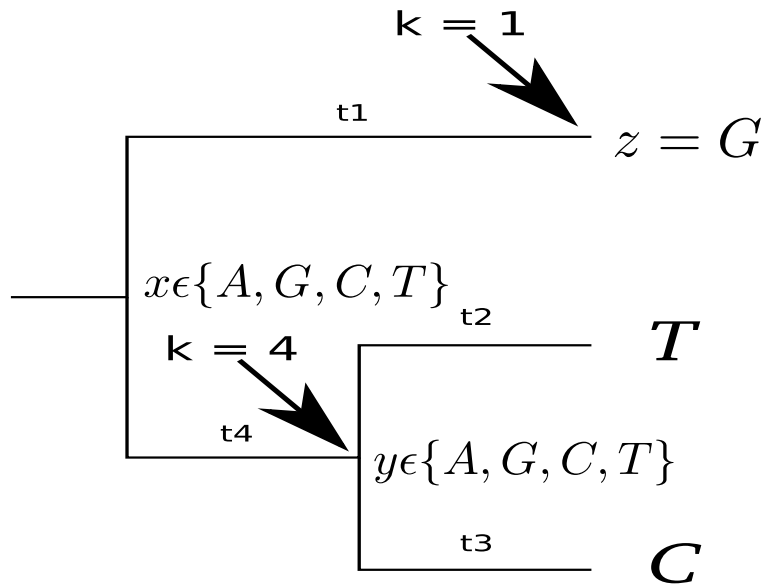


Figure 2.10: $L_0^0(x) = (\sum_y P(y|x, t_4)L_4^0(y))(\sum_z P(z|x, t_1)L_1^0(z))$

in Section 3.4.2.

We shall discuss the computational complexity of Felsenstein’s algorithm and how to estimate its run time and memory footprint in detail in section 3.1.1.

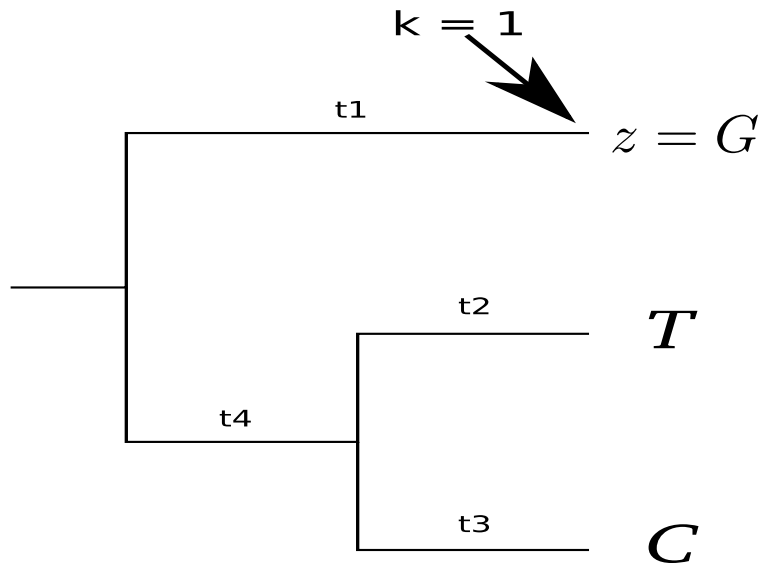
Previous work on parallelisation of Felsenstein’s algorithm used *data decomposition to segment the sequence alignment over sites*. Our main body of work is concerned with finding a suitable way to perform parallelisation of Felsenstein’s algorithm by segmenting the phylogenetic tree into subtrees. Stated differently, parallelisation is performed by performing *data decomposition over the phylogeny*. Chapter 3 discusses the parallelisation of this algorithm.

2.3 Parallel programming

In this section we give a brief overview of two concepts central to our work. Firstly, Section 2.3.1 covers the parallel programming concepts of decomposition and distribution. Then, in Section 2.3.2, we discuss the topic of parallel performance.

2.3.1 Decomposition and distribution

The concept of divide and conquer has a long history in computer science. Many theories, programming languages, and software design paradigms have

Figure 2.11: $k = 1$ is a leaf node

been built on modularising problems so that they can be more easily understood and be easier to work with. *Parallel programming (or Concurrent programming)* is one such concept, where problems are divided into smaller problems so that they can be dealt with concurrently by multiple computer processors. In this context, modularisation is called *parallelisation*. It turns out that many computational problems have properties that can be viewed as independent, and can therefore be modularised to benefit from parallelisation. Any parallel algorithm consists of two parts: a *job decomposition algorithm*, and a *job distribution algorithm* [12]. A *job* is viewed as a piece of work (i.e., a set of instructions) that has to be performed.

Job decomposition entails the manner in which the algorithm is divided into jobs. Generally speaking, job decomposition can be divided into *data decomposition* and *task decomposition* [12, 34]. Data decomposition can be done when the input data set can be grouped into independent subsets of data. The same processing that would have been done on the whole data set is then done concurrently on the individual subsets. Both the methods we discuss in this thesis, parallelisation over sites and parallelisation over the tree, are classified as data decomposition. Task decomposition can be done when parts of the algorithm can be done independently. In this case, tasks are grouped into subsets and the subsets are run concurrently.

Job distribution describes the process that is performed once the jobs have been produced by the job decomposition algorithm. There are many job distribution methods, one of which is chosen according to the input data, the

algorithm(s) used for computation, and the architecture (hardware and software) that it needs to run on.

Consider the following example to illustrate job decomposition and job distribution. Suppose there is a list $\mathcal{L} = [1, 2, 3, 4]$ and some function $f(l_i)$ that must be performed on every element of the list. Both decomposition and distribution can be done if a simple loop over \mathcal{L} is used. Each iteration takes one element of \mathcal{L} and sends it to a process p_i that will perform $f(l_i)$. Sometimes, such as our case with trees, this is not possible and two different algorithms are needed to perform the work, as Chapter 3 will show.

2.3.2 Parallel Performance

Parallel performance is measured in the *speedup* that is obtained by the parallelisation of an algorithm [13]. The relative speedup given p processors can be calculated by

$$S(p) = \frac{T(1)}{T(p)}, \quad (2.3.1)$$

where $T(1)$ is the time the algorithm runs on one processor, and $T(p)$ is the time the distributed algorithm runs on p processors. Linear speedup, $S(p) = p$, is the best-case scenario we can hope for when running code in parallel. In usual cases, relatively high speedup will be observed up to a certain number of p , by which time overhead (like communication between processes) degenerates any further gains obtained, and the speedup reaches a plateau. Super-linear speedup can sometimes be obtained, but in very rare cases. This is, usually, due to some hardware influence such as the efficient use of caching.

In most cases, there are parts of an algorithm that cannot be done in parallel. These sequential parts are not described by Equation (2.3.1). Thus, it is necessary to consider which part of the program must run sequentially and which parts can be made to run in parallel. The *sequential fraction* γ is given by

$$\gamma = \frac{T_{setup} + T_{finalisation}}{T(1)}, \quad (2.3.2)$$

where T_{setup} is the pre-computation needed before the parallel computation, and $T_{finalisation}$ is the post-computation part needed after the parallel computation is completed. The *parallel fraction* is thus $1 - \gamma$. Therefore, the total computation time with p processes, assuming the theoretical maximum of linear speedup, can be written as:

$$T(p) = \gamma T(1) + \frac{(1 - \gamma) T(1)}{p}. \quad (2.3.3)$$

We can now rewrite Equation (2.3.1) in terms of Equation (2.3.3). This gives us Amdahl's law [35, 12], which predicts the *theoretical maximum speedup* using p processors:

$$\begin{aligned}
 S(p) &= \frac{T(1)}{T(p)} \\
 &= \frac{T(1)}{\gamma T(1) + \frac{(1-\gamma) T(1)}{P}} \\
 &= \frac{T(1)}{(\gamma + \frac{(1-\gamma)}{P}) T(1)} \\
 &= \frac{1}{\gamma + \frac{1-\gamma}{p}}
 \end{aligned} \tag{2.3.4}$$

Equation (2.3.4) describes the speedup of a parallel program if there is no overhead, such as network overhead, on a computer cluster. This equation is used as an approximation of the speedup when analysis is performed, since overheads will almost always be present. It is important to consider Amdahl's law when deciding on whether to implement a part of a program in parallel or not.

2.4 Cache and memory

The memory footprint of any computation problem has a big effect on how well the problem will run on a computer, if at all. Typical modern computer architecture is based on the von Neumann model, which states that a computer has three necessary components, namely a *central processing unit (CPU)*, *memory*, and *input/output (IO)* devices [36]. Modern CPUs themselves conform to the

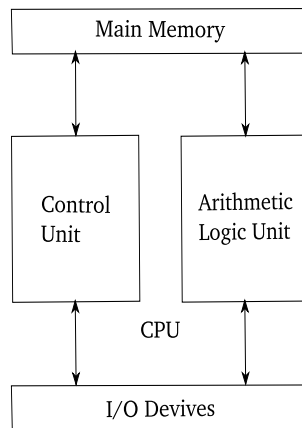


Figure 2.12: The von Neumann architecture shows the CPU that is made up by the control unit and arithmetic logic unit, main memory, and IO devices.

von Neumann model, since they have built-in memory called *CPU cache*, and IO buses [37]. We shall just refer to CPU cache as cache for brevity, but note that other caches exist, such as page cache and web cache. Cache is the second level of memory in the memory hierarchy (registers being the first, but this is irrelevant to our discussion). Cache is followed by system memory, usually random access memory (RAM), and then followed by disk drives. Cache has the fastest access time of these three levels. As we move from lower to higher levels in the memory hierarchy (for example from cache to RAM), data access times increase dramatically. A typical CPU will have more than one level of cache, with the current standard in personal computers and servers being 3 cache levels. We refer to them as *level 1 (L1)*, *level 2 (L2)*, and *level 3 (L3)* cache. The access and write times increase as the levels rise, similar to the speed difference from cache through to disks. It is, therefore, necessary for any computationally demanding algorithm to minimise memory access time as much as possible.

There are many different hardware architectures for cache, RAM, and disk drives found in computers today. The software algorithms used to manage these hardware components differ vastly between operating systems, and even between different releases of operating system kernels. Therefore, we shall not cover either cache or RAM in detail, nor make assumptions about what architectures are used by the machines on which our software is run. We shall introduce some assumptions, which are not always strictly true on specific hardware, in an attempt to construct a generic view of these components.

There are attributes that all implementations of cache and RAM share. Both have their total address space broken down into blocks. In cache, these blocks are usually referred to as *cache lines* [38]. For example, the *Quad-Core AMD Opteron Model 2380*⁴ has a L1 cache size of 128 KB partitioned into cache lines of 64 bytes each, giving a total of 2048 cache lines. The sizes of a cache line may differ between levels, with higher levels having larger cache lines than lower levels (if they differ). The size also differs between CPU architectures. In main memory, the blocks are referred to as memory blocks or pages, with the block size being the same as the cache line of the CPU's highest cache level. For our Opteron Model 2380 example, the L3 cache line size is 128 bytes [39, 38].

If the memory footprint of a data object cannot fit into one cache line, it has to be split up over multiple cache lines. It is often the case that not enough cache lines are available to store the whole data object in cache. If that is the case, a *cache miss* occurs when the processor attempts to access a memory address in the data object that is not stored inside any of the cache lines. Every time a cache miss occurs the memory page containing the memory

⁴This is the processor model used in many of the computers in the cluster we used at the *Viral Evolution Group at the School of Medicine at University of California, San Diego (UCSD)*.

address the processor is looking for must be fetched and placed in a cache line. This fetching and swapping is called *cache swapping*. Cache swapping occurs between different levels of cache as well as between the topmost level of cache and memory. So, if some data cannot be found in L1 cache, a miss occurs and L2 is searched for that data. If that data is not in L2, a miss occurs and the next level of cache or memory is searched. This process continues up to the highest level of the memory hierarchy. At that stage, it will be propagated back down the memory hierarchy.

We shall now give an example to illustrate why cache misses and swapping are important in our work. Assume that Q is a rate matrix of a codon model. Since this is a 61×61 matrix of real numbers (the double data type is used, which is 8 bytes long on standard systems), Q has a memory footprint of $61 \times 61 \times 8 = 29768$ bytes. As stated previously, a cache line has 64 bytes. This means Q will need to occupy $\lceil 29768/64 \rceil = 466$ cache lines in L1 cache. Therefore, the processor will have to perform 466 cache swaps to put the whole rate matrix in cache if Q is not stored in cache already. This cost becomes important if Q is to be loaded into cache repeatedly, as we shall see in Section 3.2. In contrast, assume that Q is a rate matrix of a nucleotide model. Then, the memory footprint of Q is only $4 \times 4 \times 8 = 128$ bytes, and only $\lceil 128/64 \rceil = 2$ cache lines are needed. Thus, there are only two cache swaps required to gain access to all the elements of Q . This clearly shows that codon models have much more overhead than nucleotide models, and that minimising the memory footprint will reduce cache misses and hence, cache swaps. We can also deduce that fewer cache misses will occur if memory addresses that are used in a sequence of process instructions are kept sequentially in memory, since, if the addresses all fit into one cache line, no cache swapping will be needed. However, if the sequential memory is larger than a cache line or memory page (depending on the level of memory we are observing), a swap will still be needed if the boundaries of the page are exceeded.

Section 3.2 will discuss the impact of cache and memory on the performance of the likelihood calculations.

2.5 Parallelisation approaches in phylogenetics

The literature on parallelisation in phylogenetics has thus far been a mix and match of various parallelisation approaches. It can be roughly grouped into two categories.

The first can be summarised as parallelisation done when inferring phylogenies, of which a number of different approaches have been followed. This is discussed in Section 2.5.1.

The second category is comprised of two approaches that perform parallelisation on the likelihood calculation. Unlike the approaches in the previous category, these approaches can be used in general phylogenetic analyses, since

they are performed on the likelihood calculation itself. Currently, the simplest of these techniques performs data decomposition over the given sequence alignment to segment the data into independent blocks that are distributed over parallel processes. We cover this approach in Section 2.5.2. Another approach uses *graphic processing units (GPUs)* to concurrently calculate the transition probability matrices used in Felsenstein's pruning algorithm. The GPU approach is covered in Section 2.5.3.

2.5.1 Parallelisation when inferring phylogenies

The first and simplest approach that has been proposed distributes the different phylogenies that are considered in phylogeny inference over multiple processors [40, 41]. This is trivial to implement, and allows for linear speedup in the number of individual processors used, which can be put to great use on computer clusters. However, this approach still struggles when computation is done for large phylogenies, which we shall discuss in Chapter 3.

Another approach runs multiple *Markov chain Monte Carlo* samplers over parallel processes when Bayesian phylogenetic inference is performed [17, 42]. This MCMC approach is not directly related to our work, but it should be noted that our approach can be used in conjunction with this approach, since they make use of Felsenstein's algorithm to perform likelihood-based assessment.

In the last decade there have been some parallelisation strategies used in phylogeny inference that are similar to our approach in that they effectively split (i.e., segment) a phylogeny into subtrees.

The first of these approaches uses the idea of inferring a phylogeny by building it with subtrees of four leaves each, known as *quartets* [43, 44, 16, 45]. All the possible arrangements of leaf nodes are evaluated by using a method such as maximum likelihood to establish which arrangement is the best for a given subtree [16]. The best subtrees are then stitched together to form the larger tree. The evaluation process of each subtree is then assigned to a parallel process. Various parallel processing strategies are contemplated in the literature.

An idea closely related to quartets was proposed by Du et al., which use a simple master/worker process architecture and data decomposition to split the data set into smaller subsets of sequences and builds subtrees for these subsets [46]. This differs from the quartets approach in that the size of the subtrees can be variable. They then use a guide tree to decompose the data set and use parallel processes to build a subtree for each subset. A master process will collect the subtrees from the worker processes and construct a phylogeny from them. This phylogeny is then considered the guide tree, and the process is repeated for a number of iterations. The guide tree obtained by the last iteration is then considered to be the reconstructed phylogeny.

In these last two cases the concept of distributing computation over subtrees is used, similar to our approach. However, they are focused solely on inferring a phylogeny given a sequence alignment. They do not address the problem of calculating a likelihood on an existing phylogeny. Our work differs from these methods in that it focuses on data decomposition for a likelihood calculation, which can be used in any implementation that uses Felsenstein's algorithm, and not just phylogeny inference.

All of these approaches obtain run time speedups as expected with parallel implementations. These implementations also obtain more accurate phylogenies than non-parallel implementations, since extra computation can be performed in the time gained by the run time speedup.

2.5.2 Parallelisation of the likelihood calculation over sequence alignment sites

Data decomposition over sequence alignment sites has been done in previous work on parallelisation of phylogenies, as stated in Section 2.2.4. In this approach a sequence alignment is segmented into alignment subsets over its sites and distributed to multiple processing cores, after which the likelihood calculation is run concurrently over the subsets [17, 18, 47, 16].

Near linear speedup increases have been recorded when using this method. Unfortunately these results come from implementations where the parallelisation over sites is only part of a bigger solution, so identifying the precise impact data decomposition over sites had in these cases is not clear from these studies.

This approach duplicates its memory footprint when the likelihood calculations for individual sites are distributed over separate processes. The matrix exponentiation, explained in Section 2.2.3, is also duplicated if each separate process recomputes the transition probability matrices instead of using inter-process communication typical to parallel programs. This is clearly wasteful, and we shall show in Section 3.2.1 that our proposed method performs better.

2.5.3 Likelihood parallelisation on GPUs

A more recent development introduced by Suchard and Rambaut uses task decomposition, where the sum-products and matrix multiplications in Felsenstein's pruning algorithm (Equation (2.2.6)) are distributed over specialised GPUs [18]. They obtained speedup factors of up to 144 for these calculations in the experiments they presented. This was made possible by the single instruction multiple data (SIMD) paradigm used in GPU design, which is designed to perform operations on matrices.

GPU solutions are very attractive due to the relatively low cost of the hardware compared to computer clusters, and most high-end notebooks and

workstations also come with GPUs built-in. However, access to dedicated GPU servers has only recently become more common in typical research environments, unlike access to computer clusters. Just like our solution, the GPU solutions also require additional specialised programming, which can be hard to integrate into existing software infrastructure.

Nevertheless, the GPU solution obtains very impressive speedup results, and can even perform well on a single CPU; most notably RAxML developed by Stamatakis that allows for inference of phylogenies with 1000 taxa in less than 24 hours [16]. It is possible to incorporate this approach with our proposed method if the CPUs in use have access to GPU processors, which will work especially well on very large phylogenies, where larger subtrees can be constructed to fill the memory of the GPUs. In Chapter 3 we shall discuss how these subtrees work.

2.6 Summary

In this chapter, we introduced the main concepts and theory that form the basis of this thesis. The concept of a likelihood function was covered in Section 2.1. The important phylogenetic topics of sequence alignments, probabilistic models of evolution, and phylogenetic trees were discussed next in Section 2.2 so that a thorough understanding of Felsenstein's pruning algorithm could be formed. Task decomposition, task distribution, and parallel performance were covered in Section 2.3. A brief introduction on CPU cache and memory was given in Section 2.4. Lastly, we covered the various parallelisation approaches that have been used in phylogenetics in Section 2.5.

Chapter 3

Parallel Likelihood Algorithms

Currently, it is common to perform phylogenetic analyses on alignments consisting of hundreds of sequences. Unfortunately, the computation of such analyses is still problematic, since run time durations can easily run into hours or even days.

In this thesis, we discuss the goal of our work, which is to decrease the computation time of phylogenetic analyses involving likelihood calculations, hence increasing the size of sequence alignments that can be analysed. We propose to obtain this decrease in computation time by modifying Felsenstein's likelihood algorithm so that parallelisation can be performed over phylogenies instead of the previously proposed methods discussed in Chapter 2: parallelisation over sites, and parallelisation of the sum-product using GPUs. Our proposed type of parallelisation is specifically meant for distributed memory systems, such as computer clusters.

In this chapter, we discuss the reasoning behind our proposed method of parallelisation, cover the theoretical aspects of how this method is performed, as well as discuss the design of the system that implements our proposed method. Section 3.1 discusses equations used to estimate the run time and memory footprint of Felsenstein's algorithm. These estimations are used when simulating the likelihood calculation, and in analyses done on data obtained from experiments we ran to investigate run times of the likelihood calculation. Section 3.2 covers the impact of the likelihood computation memory footprint on the computation time of the algorithm. We explain how keeping the memory footprint as low as possible lowers the amount of memory swapping that is needed, which, in effect, improves an algorithm's run time performance. Section 3.3 covers how we estimated the run times and built a database of run times for use in the system that implements our parallelisation method. Lastly, Section 3.4 discusses the various parts of our proposed parallelisation system.

3.1 Complexity analyses of Felsenstein's algorithm

Section 2.2.4 noted that the time complexity of the likelihood calculation is linear in the number of sequences, $O(n)$. In this section, we analyse the computational complexity of the algorithm in more detail to establish equations that we shall use to estimate the run time and memory footprint of the algorithm. These estimations are used in simulations of our parallelisation method used for analyses.

3.1.1 Estimating the run time

To analyse the computational complexity of Felsenstein's algorithm, the algorithm can be viewed to have three important parts: the sum-product when calculating the inner nodes, rate matrix exponentiation for each branch, and accessing the sequence alignment once a leaf node is reached. The computation time is a function of the number of characters in the alphabet N , the number of unique *site patterns* of the alignment m and the number of sequences in the alignment n . We refer to an alignment column (i.e., the characters for the different sequences at a single site) as a site pattern [48]. The likelihood calculation is run only once per site pattern to avoid redundant computation. Therefore, the number of site patterns is counted as opposed to the number of sites.

We cannot predict the exact manner in which the three terms are implemented, and, hence, the number of instructions that will be run. Thus, for each term we introduce one constant which represents the computation time needed to perform the computations $\mathbf{c} = [c_0 \ c_1 \ c_2]$.

The computation time of the inner nodes in Felsenstein's algorithm is

$$\psi_0(N, m, n) = c_0 N^2 m (n - 1). \quad (3.1.1)$$

This is clear if the quantity $\prod_i (\sum_y P(y|x, t_i) L_i^u(y))$ in Equation (2.2.6) is analysed. There are $n - 1$ inner nodes for a rooted bifurcating tree (as covered in Section 2.2.2), and there are N possible characters at each node. Thus, there are N possible character values at each inner node k , and N possible character values at each child node of k . Therefore, there are N^2 calculations done for $n - 1$ nodes. This will be done for each unique site pattern of the sequence alignment which is m times. In conclusion, N^2 calculations of time c_0 are performed for $n - 1$ nodes at m sites, giving Equation (3.1.1).

The computation time of rate matrix exponentiation is

$$\psi_1(N, n) = c_1 N^3 (2n - 2). \quad (3.1.2)$$

In the worst case each branch will have a different branch length. Generally, it is the case that most of the branches have different branch lengths. Hence,

in the worst case a transition probability matrix must be computed at each branch, so Equation (2.2.2) is done $2n - 2$ times. Note that the same matrix $P(t_i)$ is used for every site, and therefore ψ_1 is not linear in m . The asymptotic complexity of the matrix exponentiation is $O(N^3)$ [49] (as was stated in Section 2.2.3). However, this assumes that element-wise multiplication is done, and no matrix-multiplication optimisation such as *Strassen's algorithm* [49, 50] is performed, which runs in $O(N^{2.808})$ [50]. Systems like HyPhy do not use algorithms such as Strassen's algorithm, since these algorithms are slower for the values of N that are encountered in this problem. This is because the increased constant factor in the run time of these algorithms dominates the decrease from N^3 to $N^{2.808}$. c_1 is the computation time of the operations needed to perform the matrix exponentiation at one of the nodes.

The computation at the leaf nodes is

$$\psi_2(N, m, n) = c_2 N m n. \quad (3.1.3)$$

The value of the character at a leaf node is fetched from the sequence alignment data for each leaf node at each site, and is then compared to each of the N possible characters. The constant c_2 is the computation time of the operations needed to perform the lookup of the character.

ψ_0 , ψ_1 and ψ_2 can be combined into one equation

$$\psi(N, m, n) = c_0 N^2 m (n - 1) + c_1 N^3 (n - 1) + c_2 N m n. \quad (3.1.4)$$

If $m \gg N$ then Equation (3.1.1) is the dominant term, if $m \ll N$ then Equation (3.1.2) is the dominant term. Equation (3.1.3) shall never be the dominant term, since we expect c_3 to be very small, and in terms of order of magnitude it is the smallest term. As we shall show later, it is important to know which term is dominant in the estimation of, and in analysing, the computation times.

3.1.2 Estimating the memory footprint

The components that contain all the information needed to form the probabilistic model θ (covered in Section 2.2.3), which is used to calculate Felsenstein's algorithm, are broken down into three distinct parts: the sequence alignment, the tree topology, and the set of rate matrices. Each has a different set of parameters dictating the size of the individual data structures. Therefore, the same approach is taken to calculate the memory footprint as was taken to calculate computation time using Equations 3.1.1 – 3.1.4.

The algorithm must traverse the tree topology of the phylogeny, thus the topology must be stored in memory. We briefly discuss three ways in which the topology can be implemented. Throughout this thesis, it is assumed that the tree is a rooted complete bifurcating tree, unless specified otherwise¹. Thus,

¹A complete tree is a tree where every internal node has exactly two children.

there are $2n - 1$ nodes of which $n - 1$ are internal nodes that all have two outgoing edges giving $2n - 2$ branches. The first representation of the tree is an *adjacency list* that consists of an array of pointers (node array), which point to other arrays of integers (descendant arrays)². Pointers on a 32bit system are stored in 4 bytes, but on a 64bit system, 8 bytes are used. We assume that the system is a 64bit system³. A node array index represents a node in the topology. Each node has a corresponding descendant array, in which the elements of the array are the descendants (children) of that node. For example, let there be a node 0 with two children, nodes 1 and 2. Node 0 is represented by the first element of the node array. The value of this element is a pointer to the descendant array of node 0, which in turn stores the node array indexes of the descendants. Let these indexes be 1 and 2. The node array values at indexes 1 and 2 then point to the descendant arrays of nodes 1 and 2. The branch length array is also stored. Each branch length is represented as a real number and is thus stored as a double. Note that the root node does not have an ancestral branch, and is therefore assigned 0. The function to calculate the size of a tree implemented as an adjacency list, in which node identifiers are stored as pointers, descendant identifiers (indexes of the node array) are stored as integers and includes a set of real valued branch lengths is

$$\begin{aligned}\phi_0(n) &= 8(2n - 1) + 4(2(n - 1)) + 8(2n - 1) \\ &= 40n - 24.\end{aligned}\tag{3.1.5}$$

The second representation of the tree is the node structure, where each tree node is implemented as a structure or object. The node has four members: an integer that stores the node index, two pointers for the branches pointing to child nodes, and a double to store the branch length. Take note that if a node is a leaf node, it still has two pointers. These are null pointers. Node indexes are stored in integers of 4 bytes. To calculate the tree as a node structure, the following function of n is used

$$\begin{aligned}\phi_0(n) &= 4(2n - 1) + 8(2(2n - 1)) + 8(2n - 1) \\ &= 56n - 28.\end{aligned}\tag{3.1.6}$$

Lastly, we investigated the implementation of the *TheTree* class in the *HyPhy* software package. It uses the node structure approach discussed previously, but adds some memory for administration variables used, such as pointers to the root node and current visited node. Note the change in constants. For 64-bits, 37 was added, and for 32-bits, 25 was added. The constants were

²This is only one representation of an adjacency list. If the tree needs to undergo modification it will be better to use linked-lists instead of arrays. Yet another example would be to use a map (hash-table) if fast lookups are needed on non-integer identifiers of nodes.

³The cluster on which we evaluated our approach, like most modern servers, is a 64bit system.

obtained by evaluating the program code. To calculate the size of a *TheTree* object, we used

$$\phi_0(n) = 56n + 9 \quad \text{for 64bit or,} \quad (3.1.7)$$

$$\phi_0(n) = 56n - 3 \quad \text{for 32bit.} \quad (3.1.8)$$

It is clear that these representations are all linear in n , and thus all have the same asymptotic complexity. Hence, choosing between the different approaches makes little difference when considering the memory footprint. Naturally, the representation can have an impact on how the tree traversal algorithm is implemented. Caution should be taken when choosing a representation of the tree topology. We shall use the HyPhy tree interpretation, since we used HyPhy to measure real run times in other experiments.

To calculate the probabilities needed for computing the likelihood one needs to store the rate matrix Q , and a transition probability matrix $P(t_i)$ for each branch that has a unique branch length. We assume the worst-case scenario in which each branch has a unique branch length, therefore there will be a probability matrix for each branch. The worst-case is not an unrealistic assumption, since the branch lengths are real-valued numbers. Elements of a matrix are real numbers, so the *double* data type which consists of 8 bytes is used. The function to calculate the memory footprints for one probability transition matrix $P(t_i)$ for every branch length t_i (first term), and the memory footprint of Q (second term) is

$$\phi_1(N, n) = 8N^2 2(n - 1) + 8N^2. \quad (3.1.9)$$

The full alignment must be stored in order to retrieve the observed characters at the leaf nodes. Each sequence of the alignment is simply stored as a string of characters, with each character stored in a *char* data type, taking 1 byte per char. Since there are n sequences and m sites in the alignment we can calculate the size of the alignment with

$$\phi_2(m, n) = mn. \quad (3.1.10)$$

Therefore, the total memory footprint for Felsenstein's algorithm is the sum of terms ϕ_0 (using Equation (3.1.7)), ϕ_1 and ϕ_2 giving:

$$\phi(N, m, n) = 8N^2(2n - 1) + mn + 56n + 1. \quad (3.1.11)$$

We use Equation (3.1.11) to calculate the memory footprint of Felsenstein's algorithm in some simulations and analysis. Equations 3.1.5 - 3.1.10 are used when analysing the influences of the different structures on the memory footprint, as seen in Section 3.2.1.

3.2 Effects of the memory footprint on the likelihood algorithm run times

The reason that memory footprints have a big effect on the run times of algorithms is rooted in the cache line and memory page sizes that we have briefly covered in Section 2.4. It was discussed that cache is broken up into equally sized segments called cache lines, and memory into memory pages. We introduced the concepts of cache misses and cache swapping, and how the different sizes of the nucleotide and codon models will influence the number of cache misses and therefore force cache swapping.

The example of the rate matrix given in Section 2.4 is relevant to our work, since rate matrix exponentiation as covered in Section 2.2.3 is one of the main terms that makes up the computation of Felsenstein's algorithm. In Felsenstein's pruning algorithm, it is usually the case that we have to compute multiple transition probability matrices $P(t_i)$, one for each unique branch length t_i . For larger phylogenies, such as those we are concerned about, the number of branches and, therefore, the number of transition probability matrices is very high.

Part of our hypothesis is that minimising the memory footprint of the problem will improve performance of the likelihood calculation. Naturally we cannot minimise the memory needed to compute the likelihood for the sequence alignment \mathcal{S} given the tree topology τ and model parameters θ without simplifying the evolutionary model or simplifying the algorithm itself. However, we can use data decomposition to distribute the memory footprint over multiple processors, thus minimising the memory footprint on a single processor.

3.2.1 Data decomposition of the likelihood problem

Our hypothesis states that the memory footprint per process is smaller when using parallelisation over the phylogeny than when using parallelisation over sites, and that the smaller memory requirement per process will decrease the computation time of the likelihood algorithm, as was stated in Section 1.3. In this section we describe these two methods of performing data decomposition on the likelihood calculation, and then we discuss the influence these methods have on memory requirements. The difference between parallelisation over sites and over the phylogeny is illustrated in Figure 3.1. Here, the *dashed vertical lines* show three possible groups of sites that are made if parallelisation over sites is performed. When viewing this parallelisation method in context of the data, we shall call this *data decomposition over sites*. The *solid horizontal lines* show four possible subtrees and their corresponding three groups of sequences that are produced if parallelisation over the phylogeny is performed. We shall call this *data decomposition over phylogeny*.

In the case of data decomposition over sites the number of sites m can be

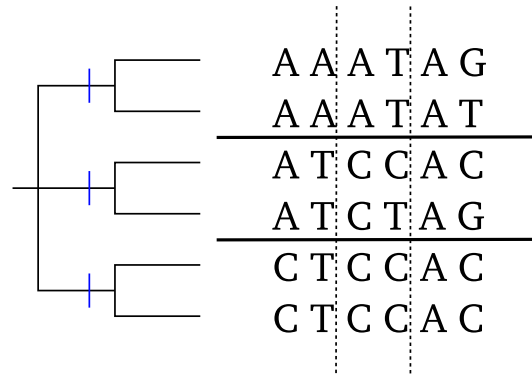


Figure 3.1: How the sequence alignment is segmented when performing data decomposition over sites and over the phylogeny.

easily divided into subsets and distributed amongst the available processes. However, in the case of data decomposition over phylogeny, it is the roughly equal-sized subtrees produced by the tree segmentation algorithm that are divided amongst the processes. The leaf nodes of subtrees are not necessarily associated with sequences, such as the inner subtree shown in Figure 3.1. Therefore, we shall specifically count the number of leaves instead of sequences. We shall use the notation n' when we refer to the number of leaves in a subtree and m' when we refer to the number of sites in a subset.

The two data decomposition methods, depicted in Figure 3.1, affect the terms in Equation (3.1.11) differently. Only Equation (3.1.10) is affected when data decomposition over sites is performed, since it is the only term that is a function of m . When data decomposition over phylogeny is performed, each process needs only the memory of the subtrees assigned to it by the process manager. This entails that the process only needs the subset of transition probability matrices $P(t_i)$ where $t_i \in \mathbf{t}$ is associated with those subtrees. This necessarily means that there will be less cache swapping of transition probability matrices over the phylogeny than over sites, since if fewer matrices are needed, it is more likely that these matrices will reside in cache, and fewer cache misses and cache swapping is needed. Therefore, Equations 3.1.7, 3.1.9 and 3.1.10 are all affected by decomposition over sequences, since they are all functions of n .

To illustrate the difference between the memory footprint of data decomposition over the phylogeny and data decomposition over sites, we use Equations 3.1.7, 3.1.9 and 3.1.10 to calculate the memory size as a function of the number of parallel processes p . The dimensions of the *H1N1 HA gene* data set from the *NCBI influenza database* are used (see Section 4.1). This sequence alignment consists of 4928 sequences that each have 1922 sites. We started with the memory footprint of only one process, which is the base case (original memory footprint), and incremented the number of processes by one up to $p = 50$. The results of these experiments are shown in Figures 3.2 and 3.3.

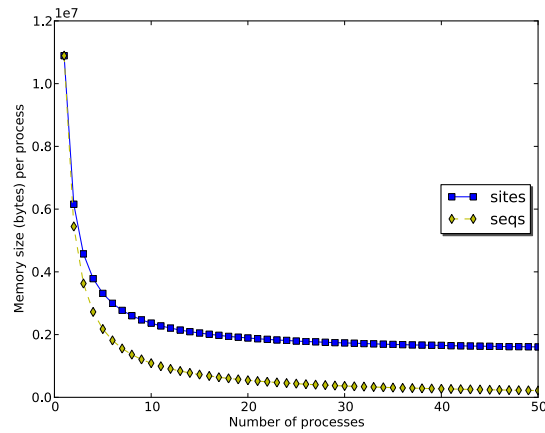


Figure 3.2: Data decomposition over sites and sequence of the H1N1 HA gene data set: memory size as a function of the number of processes using a nucleotide model.

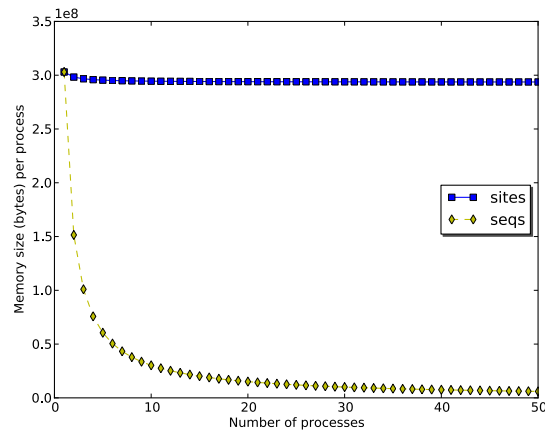


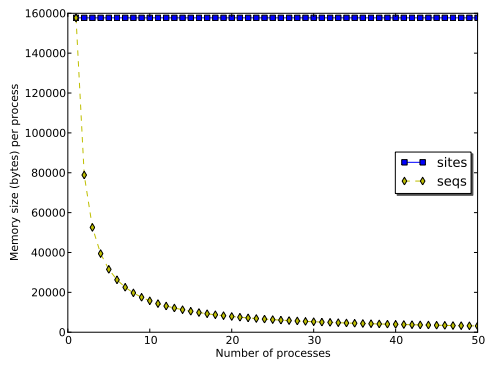
Figure 3.3: Figure 3.2 repeated for a codon model.

The values plotted show the memory footprint of *one process* after data decomposition was completed using p processes. Figure 3.2 shows the footprint when a nucleotide model is used. Here, a dramatic initial decrease is seen for both data decomposition over sites and data decomposition over the phylogeny, with data decomposition over the phylogeny being more effective. However, when a codon model is used, such as in Figure 3.3, data decomposition over the phylogeny completely outperforms data decomposition over sites, which shows almost no decrease in the size of the memory footprint.

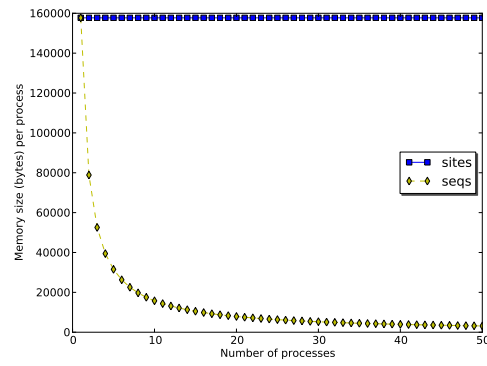
Using the same data set as before, we calculated the memory footprints of the tree, model (matrices) and alignment structures individually as a function of p . Figure 3.4 shows the results for these experiments. The results for the

tree and model footprints are shown in Figures 3.4a - 3.4d. We see the expected dramatic decrease in size when data decomposition over the phylogeny is used, since the subtrees only contain parts of the original tree and a subset of matrices associated with the branches of these subtrees. Figures 3.4e - 3.4f shows how the memory footprint of alignment is affected. In both cases, the operations to access the characters of the sequences are divided amongst p processes, and, therefore, we see that the memory footprints are the same.

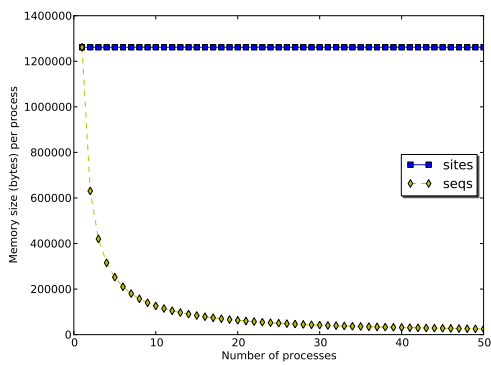
Through these examples, we have shown that performing data decomposition over the phylogeny reduces the memory footprint of Felsenstein's algorithm dramatically, regardless of which probabilistic evolutionary model is used. Thus, since the memory footprint is decreased, it is also more likely that the number of cache misses decreased. Therefore, as discussed in Section 2.4, the computation time of the likelihood calculation decreases.



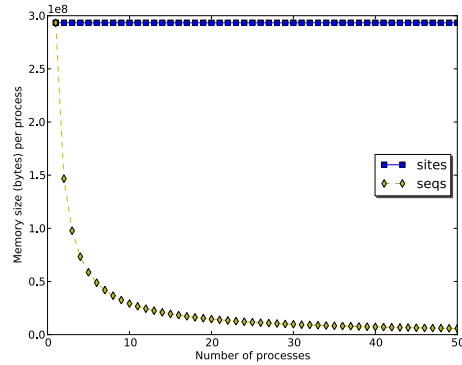
(a) Nucleotide model, tree footprint



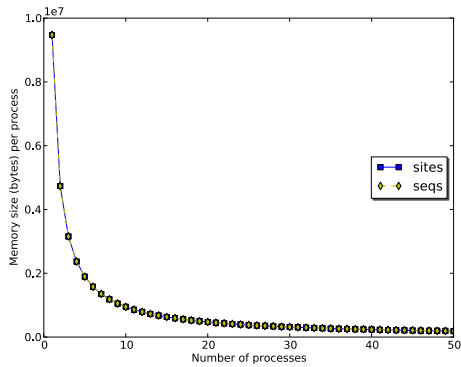
(b) Codon model, tree footprint



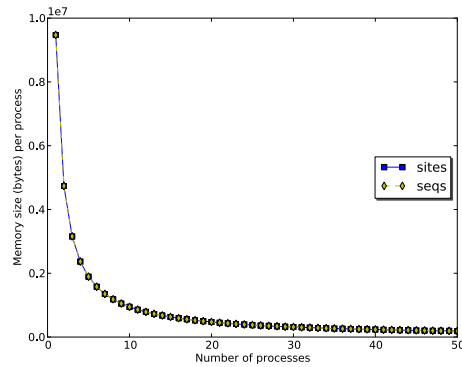
(c) Nucleotide model, Matrices footprint



(d) Codon model, Matrices footprint



(e) Nucleotide model, Alignment footprint



(f) Codon model, Alignment footprint

Figure 3.4: Data decomposition performed individually for the tree, model and alignment structures, over sites and sequence of the H1N1 HA gene data set.

3.2.2 Likelihood calculation run times and cache limits

Data decomposition over the phylogeny drastically decreases the memory footprint per process, as shown in Section 3.2.1. Nevertheless, it is still a priority to limit the unnecessary waste of processor cycles which results from excessive cache swapping done on larger phylogenies. To this end, we designed an experiment to record the run times of likelihood calculations given various phylogeny sizes in order to find a theoretical optimum. We hypothesised that through comparing the run times one will be able to find the optimal memory footprint size, which we shall call the *cache limit*. The cache limit is an amount of memory; exceeding the cache limit results in unnecessary cache swapping. Therefore, our goal was to find the *optimal subtree size* by finding the maximum memory footprint size that does not exceed the cache limit.

In order to find this optimal subtree size, we must profile the likelihood calculation run time on multiple phylogeny sizes. We used an installation of the HyPhy package installed on a high-performance computing (HPC) cluster to compute the likelihoods. Then, we developed profiling code to measure the time that expires from the moment the likelihood calculation is started until the moment it is completed. All the other processing time incurred by the program is irrelevant to Felsenstein's algorithm, and is, therefore, ignored. The likelihood value is also ignored, since we are only interested in the time it took to compute the likelihood. Inaccuracies can occur during profiling due to unforeseen events that can influence the likelihood calculation such as network latency and switching costs incurred by workload balancing when the *operating system (OS)* switches between processes on the CPU. Thus, to increase the profiling accuracy of a given phylogeny size, the calculation is run multiple times and the mean run time is computed.

The experiment was implemented using the popular *Message Passing Interface (MPI)* API so that multiple likelihood calculations could be performed concurrently on multiple processors for the same phylogeny sizes. This allowed us to increase the number of likelihood calculations that could be profiled. We ran one hundred likelihood calculations (ten times on ten processes) for each phylogeny size from two to 100. For each likelihood calculation, a random sequence alignment with the desired dimensions was generated, a phylogeny was inferred and the likelihood calculation was run and profiled. For each phylogeny size, the mean of all recorded run times was computed. This was done for both nucleotide and codon models. For both models, three experiments were ran. Each experiment has a constant number of sites: 300, 1200, and 3000.

The results of the experiments are shown in Figures 3.5 through 3.7. Figure 3.5 shows how the mean run times increased as the phylogeny sizes increased when nucleotide models were used. For clarity, Figure 3.6 shows these results up to 20 sequences: in this region, larger than linear increases are observed. Figure 3.7 shows the linear increase in mean run times for tested

phylogeny sizes when codon models were used.

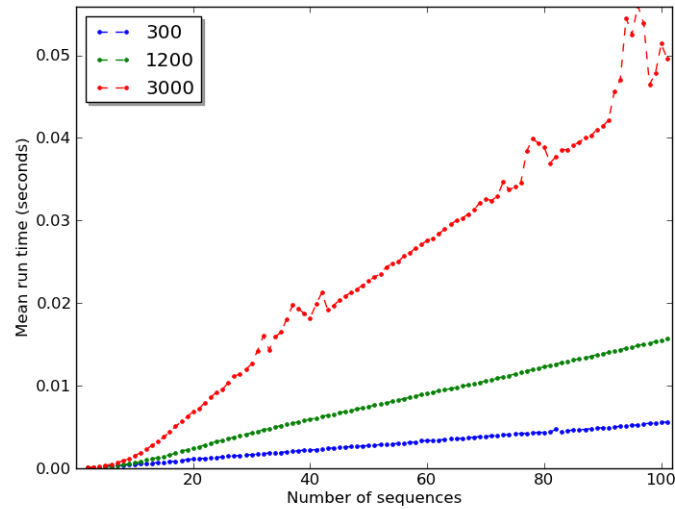


Figure 3.5: HyPhy runs performed using nucleotide models on the cluster at the Viral Evolution Group at the School of Medicine UCSD.

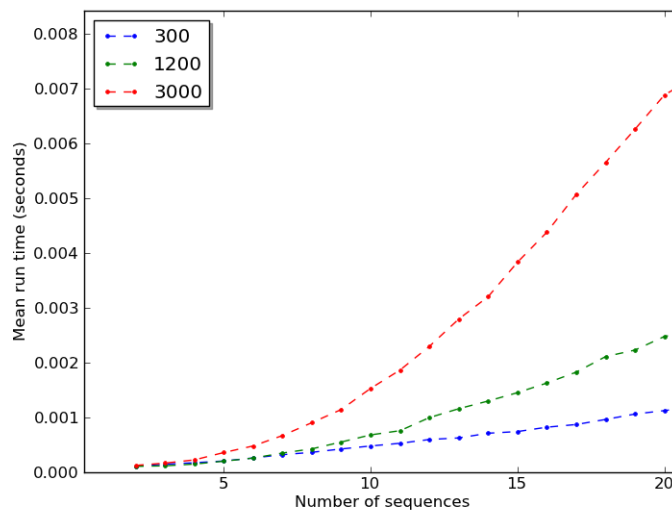


Figure 3.6: Figure 3.5 zoomed in: showing the non-linear increase in run time observed for the smaller phylogeny sizes when nucleotide models are used.

In these experiments we expected to see a linear increase in run times up to the cache limits, since the run time Equation (3.1.4) is linear in n for a

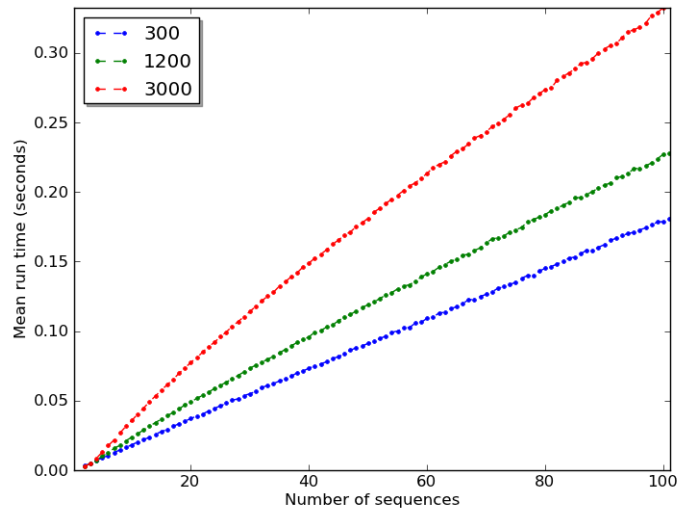


Figure 3.7: HyPhy runs performed using codon models on the cluster at the Viral Evolution Group at the School of Medicine UCSD.

constant m and N . A drastic increase in run times will be seen once the cache limits are reached. Tables 3.1 and 3.2 show the number of sequences and sites theoretically needed to fill L1 and L2 cache when nucleotide and codon models are used respectively. However, the memory footprints for phylogenies of these sizes will not actually fit into cache, since the cache has to store other data as well. For example, the operating system instructions need memory to store data relevant to their operation. We observe this linear increase up until the cache limit has been exceeded when using nucleotide models (Figure 3.5), although the sizes are much smaller than expected: $n = 20$ for $m = 300$, $n = 9$ for $m = 1200$, and $n = 5$ for $m = 3000$. For codon models in Figure 3.7, we observe a linear increase in run times, but not the drastic increase in run times when the cache limit was reached, as was the case for nucleotide models. This is due to the cache limit already being exceeded at, or very close to, $n = 2$ when codon models are used. This is corroborated by Table 3.2, which shows that no phylogenies will fit into L1 cache for any of the selected values of m .

# sites	# seqs in L1	# seqs in L2
300	110	879
1200	43	350
3000	19	158

Table 3.1: The maximum number of sequences that can fit into L1 and L2 when using nucleotide models for: $m = 300$, $m = 1200$, and $m = 3000$.

# sites	# seqs in L1	# seqs in L2
300	0	19
1200	0	9
3000	0	8

Table 3.2: The maximum number of sequences that can fit into L1 and L2 cache when using codons models for: $m = 300$, $m = 1200$ and $m = 3000$.

Our algorithm exploits the increase in run times shown in the above figures. For example, if nucleotide models are used, the run time of the likelihood calculation is 51.4 milliseconds for a data set where $n = 100$ and $m = 3000$, but only 1.53 milliseconds for a data set where $n = 10$ and $m = 3000$. This is an increase of a factor 33.5 from $n = 10$ to $n = 100$. Therefore, if the phylogeny is segmented into smaller subtrees and their likelihoods are calculated concurrently, we shall obtain a speedup (Section 2.3.2). This illustrates our key motivation for performing parallelisation over the phylogeny, which shows that the run times of the smaller phylogenies are much less than the larger phylogenies, thus leading to the algorithm we developed to perform this data decomposition. We shall discuss our algorithm that exploits this drastic difference in run times in following sections.

3.3 Estimating the likelihood calculation run times

Section 3.4 covers our proposed system, which finds the best segmentation of a given phylogeny, so that the resulting subtrees can be distributed over parallel processes. In order to do this, the system must estimate the run times of the likelihood calculations on the original phylogeny and subtrees. We estimated the likelihood calculation run times (or, for brevity, run time estimations) in three ways.

The first method uses Equation (3.1.11) to obtain the run time for a given phylogeny size. However, the results given by the equation does not take hardware overheads into account, and, therefore, we developed more accurate ways of estimating the likelihood calculation run time of a phylogeny.

The second method uses recorded run times of various phylogeny sizes. The run times were obtained for given phylogeny sizes by performing likelihood calculations on phylogenies with the desired sizes, using the HyPhy package. HyPhy was slightly modified for this purpose, by incorporating profiling code, which measured the duration of the likelihood calculation. For example, the run time of a phylogeny, where $n = 10$ and $m = 300$, was obtained by running the likelihood calculation and observing the result of the profiling. These profiled likelihood calculation run times were then recorded and stored in a

database, which is used each time the system requires a run time estimate of a given subtree. The database was initialised with the run times obtained by the experiments that were presented in Section 3.2.2. Thereafter, the database was updated in the same manner, with new run times selected from the previous experiments, which are discussed in Chapter 4.

The third method performs linear regression on previously recorded run times to estimate the likelihood calculation run time. This method can be used when the desired phylogeny size is not in the run times database, instead of running the likelihood calculation on the phylogeny to record the run time for future use. However, multiple run times of an appropriate size are needed to perform regression; we call these run times data points. The data points used when performing regression (i.e., the phylogeny sizes and their corresponding run times) should have roughly the same number of sites, but stretch over a range of number of sequences, for example, $n = 2$ to $n = 100$ in iterations of ten. This means that regression cannot be run if there is a lack of appropriate data points. Therefore, the experiments performed in Section 4 did not use regression, since the database had not been populated with the appropriate data. Regression estimates the run times rather poorly, even if appropriate data were available. The poor performance is especially noticeable when nucleotide models are used, since these run times do not increase linearly, as can be seen in Figure 3.5.

The recorded run times method was used in our proposed system, since this method obtains higher accuracy (i.e., real run time measurements) than both the computational complexity and regression methods.

3.4 Proposed system overview

As we've seen from the arguments in the previous sections, it is theoretically advantageous to perform parallelisation over phylogenies. We, therefore, propose that any given tree topology is segmented into subtrees. In order to do this, the size of the subtrees that that results in the fastest run times has to be established before a likelihood can be calculated. We call this the optimal subtree size. Therefore, in this Section, we propose a preprocessing system that finds the theoretical optimal run times.

Figure 3.8 is a flowchart that shows the basic workings of the algorithm proposed for the preprocessing step. The system takes the tree of interest as input. Then, a threshold is chosen, which is used to help determine the optimal size of the subtrees (Section 3.4.1). The threshold is then used to perform *tree segmentation* that results in a set of subtrees. We developed the *phylogenetic tree segmentation algorithm (PTS)* for this purpose (Section 3.4.2). Thereafter, the subtrees are distributed amongst available processes by the *process manager* using *process queues* to prioritise the subtrees assigned to each process (Section 3.4.3). Using the process queues, the run time, given the

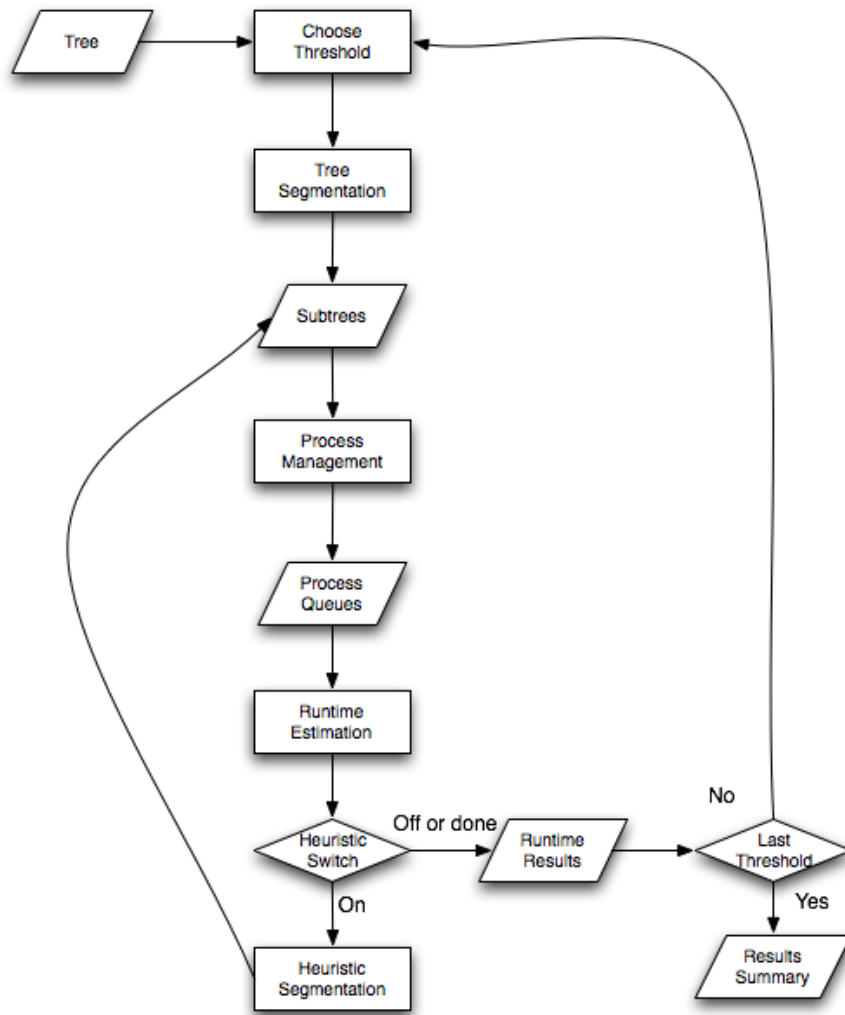


Figure 3.8: Flowchart showing an overview of our proposed parallel likelihood algorithm.

chosen threshold, is estimated by using the recorded likelihood calculation run times in the run time database when possible. The run time database and how it is populated was discussed in Section 3.3. Section 3.4.4 describes how the proposed system use the run time estimations to predict the best segmentation scheme. The run time estimation may then be optimised by the *heuristic segmentation algorithm*, which was developed as an added refinement to the standard segmentation algorithm (Section 3.4.5). Note that this heuristic is optional, and other heuristics can be developed and used at this step. The heuristic segmentation algorithm produces a new set of subtrees that must be reprocessed by the process manager, after which the run time estimation is recalculated. Afterwards, the run time results are analysed, and the system decides whether the run time can be improved upon. If this is the case a new

threshold is chosen and the process is repeated. This is done until the system deems the run time to be optimal.

The proposed system was implemented in a set of tools that we used throughout development, testing and verification of our work. We shall henceforth refer to this software as *the simulator*.

In the rest of the section, we shall discuss each of the parts of the proposed system in detail.

3.4.1 Finding the optimal threshold

In Section 3.2, we stated that the desired size of the subtrees is the size that will allow for the optimal memory footprint, which, in turn, should result in the optimal run time. In our system, the desired subtree size of n' can be obtained once the input phylogenetic tree of size n is given to the system. We call this desired subtree size the *threshold*. The threshold is chosen by selecting the optimal result from multiple tested possibilities.

In our simulator we implemented two approaches for choosing a threshold. Both cases are iterations from a minimum to a maximum threshold. However, they differ in their termination conditions. In the first case, the value of the limit λ is set equal to the height of the tree, and the loop runs until $n' = \lambda$. This has the effect that each possible threshold value is tested. In the second case, the value of λ is set to some positive value smaller than the height of the tree. Once again, the loop stops when $n' = \lambda$. This was done since we hypothesised that an optimal threshold will not be close to the height of the tree. Thresholds of up to roughly $n' = 20$ for nucleotides and $n' = 5$ for codons were suggested based on the experiments done in Section 3.2.2. Using the first case, the run time of the segmentation algorithm in the simulator was negligible (4.928 seconds for $n = 2000$ and $m = 3000$) compared to the total run time spent during likelihood calculations (1334.67 seconds for $n = 2000$ and $m = 3000$). Therefore, it was concluded that the first case could be used if there is any doubt as to what the value of λ must be for a particular phylogeny.

Our experiments showed that multiple values of n' can often produce the same run time. We, therefore, propose that the largest value of n' is chosen as the threshold, if multiple values of n' produces the optimal run time. This is useful in reducing communication overhead between parallel processes, as we shall discuss in Sections 3.4.3 and 3.4.3.1.

3.4.2 Phylogenetic tree segmentation algorithm (PTS)

The system can segment the original phylogeny into subtrees of size n' once the optimal threshold has been obtained. These segmentations are performed by the *phylogenetic tree segmentation algorithm (PTS)*. Our algorithm uses tree traversal to identify the nodes at which the phylogeny must be segmented into a set of subtrees $\kappa = \{\kappa_0, \kappa_1, \dots, \kappa_n\}$. The segmentation process is explained in

Section 3.4.2.1. Segmenting the phylogeny introduces *dependencies*, where, for example, a subtree κ_i is dependent on κ_j , in which case the likelihood of κ_i cannot be calculated before the likelihood of κ_j is calculated. Any subtree κ_i that has dependencies is assigned a *substitution vector* for each dependency, where the substitution vector stores the probabilities needed to compute the likelihood of κ_i . We cover dependencies and substitution vectors in Section 3.4.2.2. Section 3.4.2.3 gives an overview of alternatives to the methods we propose.

3.4.2.1 Segmenting trees

PTS, which was inspired by Felsenstein's algorithm, uses *post-order traversal* to traverse the tree bottom-up and record the number of visited nodes in order to find the desired subtree root nodes. Our algorithm is able to record the number of descendants that each node possesses, since post-order traversal visits child nodes before parent nodes. Thus, when a parent node k is visited, the total number of descendants have already been recorded, and, therefore, the algorithm will know if the desired size of the subtree n' has been reached. Node k is identified as the root node of the subtree if n' has been reached when k is visited. It is then at k where the phylogeny will be segmented to obtain the subtree κ_k . Otherwise, if n' has not been reached at k , the traversal algorithm continues the search. Note that the other traversal methods, pre-order, in-order and level-order traversal, will not work for our purposes, since they are top-down instead of bottom-up traversal algorithms. With a top-down traversal algorithm the ancestors are visited before the descendants. Therefore the number of descendants cannot be recorded beforehand.

The Pseudo code for PST is given in Algorithm (3.1). In the algorithm, k is the number (name) of the node currently visited, i is an immediate descendant of k , c is the number of descendants and n' is the threshold. $\text{PTS}(k)$ runs recursively over every node in the tree. The termination condition is when k is a leaf node, and since only the leaf node was observed the function returns 1. Thus, if k is not a leaf node, the function is called for every child k_i of k , and the number of descendants of k is computed, assigned to c , and returned. This is seen at lines 6 and 7. A new root node k is recorded, if c is equal to or larger than the specified threshold n' , as seen at line 9. A new root node is also recorded if k is the root node of the phylogeny, since the traversal cannot continue beyond the original root node, after which the algorithm stops by default. In both these cases, k is stored and 1 is returned. We return 1 since k is the root node of the subtree κ_k , but also a leaf node of the parent subtree of κ_k . If k is not identified as a root node, the number of descendants of k is returned as c .

We shall now follow with an illustration of how the PTS algorithm segments the phylogeny with an example using Figure 3.9. Let the desired number of nodes in a subtree be three nodes. Then the desired size of the subtree is two (i.e., $3 = 2n' - 1$, if $n' = 2$). The algorithm starts at the root node,

Algorithm 3.1 Phylogenetic tree segmentation algorithm

```

1: function PTS( $k$ )
2:   if  $k$  is a leaf node then
3:     return 1
4:   else
5:      $c := 0$ 
6:     for all  $k_i$  of  $k$  do
7:        $c := c + \text{PTS}(k_i)$ 
8:     end for
9:     if  $c \geq n'$  or  $k$  is the root node then
10:      addSubRoots( $k$ )
11:      return 1
12:    else
13:      return  $c$ 
14:    end if
15:  end if
16: end function

```

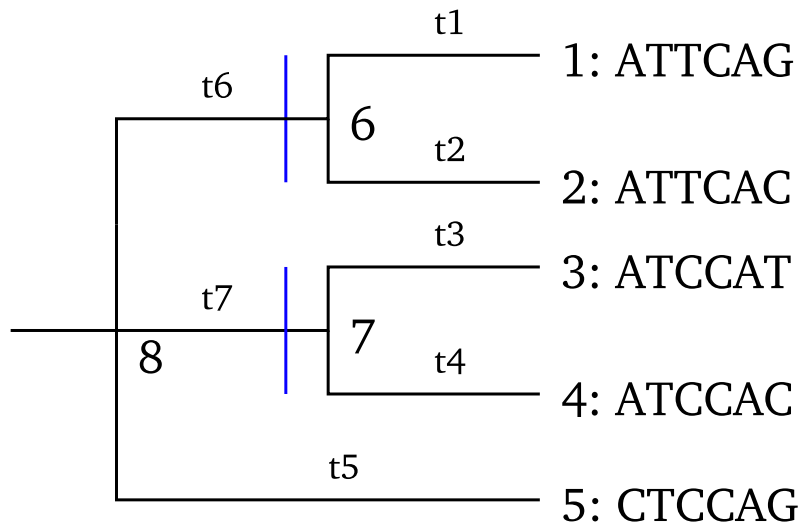


Figure 3.9: Illustration of the segmentation of this simple phylogeny into three subtrees when $n' = 2$. The original tree has a size of five, since it has five leaf nodes. The algorithm produces three subtrees, two of which have a size of two while the remaining one has a size of three.

where $k = 8$, and performs post-order traversal to traverse the topology. The termination condition is met when a leaf node is reached, for example, in the cases of nodes 1 and 2. The PTS algorithm starts counting nodes by returning one at a leaf node. After 1 and 2 have been counted, the algorithm proceeds back (since PTS is bottom-up) to the ancestor of these nodes, in this case node 6. The number of descendants of 6 are now summed, in the case $c = 2$,

which is the desired size. Since $c = n' = 2$, the decision is made to segment the tree at node 6, making node 6 the root node of the new subtree κ_6 . The same process is repeated to obtain the subtree κ_7 . The remaining subtree κ_8 has the original root node, node 8, as its root node. Note how this subtree has a size of $n' = 3$, and not $n' = 2$ as we specified. This last subtree is thus an example of where $c = 3 \geq n'$ (i.e., is greater than the threshold). It is clearly not possible to create a subtree that is equal to the desired size, since node 8 has three leaves.

Like Felsenstein's algorithm, PTS is a bottom-up dynamic programming algorithm. The sub-problems in PTS are to identify the subtrees by counting the number of nodes that should be in the subtrees. Thus, the smallest of the sub-problems is to count the number of nodes in a subtree that possess only leaf nodes. The next smallest of the sub-problems are those subtrees which have root nodes which are ancestors of leaf nodes. In this case, the subtrees will consist of three nodes if the phylogeny is bifurcating. For multifurcating phylogenies the number of nodes per subtree can vary. Thereafter, come sub-problems where a subtree has a root node that is an ancestor of the preceding subtree's root node, and so on. This means that when node k is visited, the number of descendants of k is already known. Therefore, the number of descendants at each descendant of k does not need to be recomputed by our algorithm. In other words, each node in the phylogeny is only visited once when post-order traversal is used.

3.4.2.2 Substitution vectors and subtree dependence

In Felsenstein's algorithm, the function $L_k^u(x)$ must be computed at each node k in the phylogeny for every possible character x in the model being used (nucleotide or codon model). Therefore, at each k , there are N probabilities being summed, since there are N possible characters x . However, a modified version of Felsenstein's algorithm must be run over all the subtrees that the PTS algorithm has produced.

This modification is needed, since Felsenstein's algorithm is interrupted at each segmentation point, and each segment must be calculated independently. The interruption has two effects on the likelihood calculation. Firstly, one cannot simply sum the probabilities of all possible values of x at the root node of a subtree as one would do at a standard root node (Equation (2.2.5)), since Felsenstein's algorithm expects N probabilities at a node and not a single probability. Similarly, one cannot simply return zero, one or a single probability depending on an expected character at a leaf node of a subtree. Thus, we need a placeholder to store the probabilities of each possible x at any node k that is a segmentation point. We call this placeholder a *substitution vector*. The substitution vector allows Felsenstein's algorithm to stop at a root node of a subtree and start again at its corresponding leaf node in the parent subtree.

Figures 3.10 and 3.11 illustrates how the subtree vector is introduced. Assume that a nucleotide model is used ($N = 4$). Therefore, a substitution vector with a dimensionality of four is needed. In Figure 3.10, the PTS algorithm

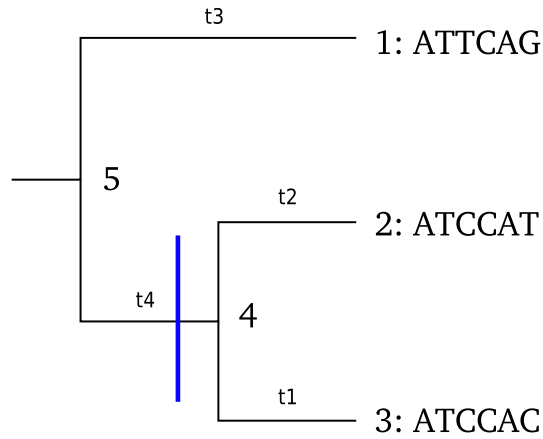


Figure 3.10: Node 4 of a simple phylogeny is chosen as the segmentation point.

chooses node 4 to be the segmentation point. In Figure 3.11, the algorithm makes the segmentation, which results in two subtrees. The first subtree has node 4 as its root node and is situated at level 2 of the original tree (i.e., one branch from the root node). The second subtree has the original root, node 1, as its root node, and its new leaf node is assigned the substitution vector with elements $[x_1 \ x_2 \ x_3 \ x_4]$.

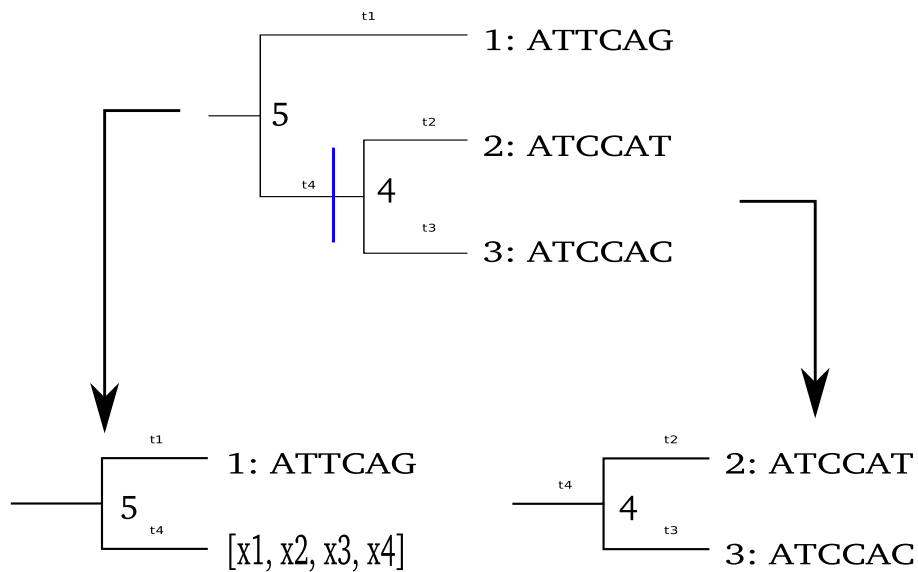


Figure 3.11: Two new subtrees are produced and a substitution vector is created.

It is clear from the above discussion that *dependencies* between subtrees are introduced at each new segmentation point. Take the phylogeny in Figure 3.10 as an example. Node 4 has been chosen as a segmentation point, and, therefore, node 4 is now the root node of the subtree κ_4 . However, node 4 is also a leaf node of the subtree κ_5 . So, if the likelihood of κ_5 is to be calculated, the probability of each possible character at node 4 must be known. Therefore, node 4 is a dependency of κ_5 . We may also refer to κ_4 as being a dependency of κ_5 .

Dependencies imply that the order in which subtree likelihood calculations are performed is important. To address this problem, the segments must be ordered (prioritised) in such a manner that the likelihood of κ_i will be calculated before the likelihood of κ_j , given that κ_i is a dependency of κ_j . This will eliminate situations where we are unable to perform a likelihood calculation due to a dependency which has not been computed.

Dependencies will also have a greater influence on run times depending on the shape of the tree topology. Our algorithm will perform better on trees that tend to be more balanced, and will perform poorer on ladder-like trees. This is due to waiting times that present a larger problem if a phylogeny is more ladder-like. In a ladder-like tree, the segmentation points will inherently be more likely to be descendants of other segmentation points. Take, for example, Figures 3.9 and 3.12. In the former subtrees κ_6 and κ_7 can be computed concurrently. However, in the latter example, this is not the case, since κ_7 is dependent on the root node of κ_6 , and must wait for the computation on κ_6 to be completed. For much larger ladder-like trees, this problem is compounded.

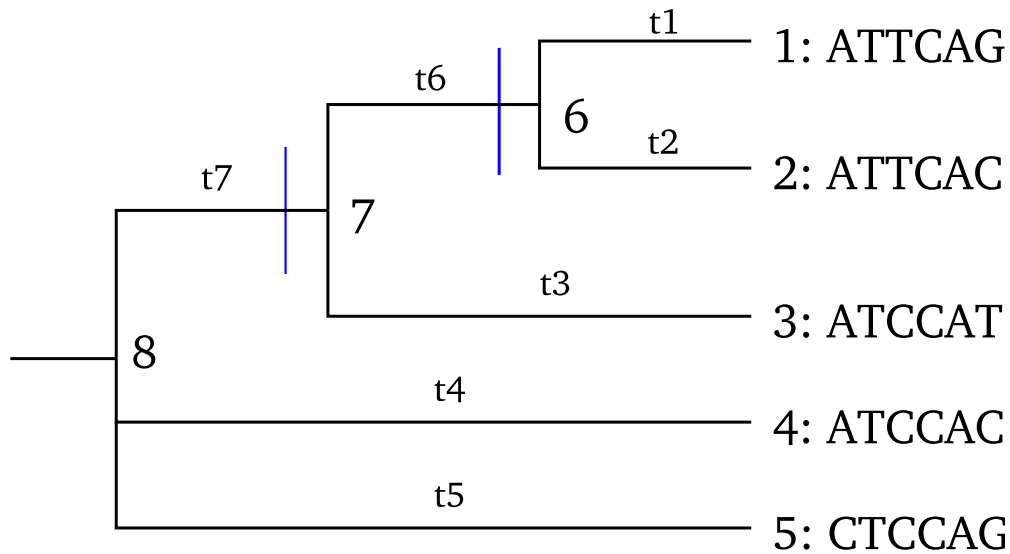


Figure 3.12: How PTS is used to segment this simple ladder-like phylogeny into three subtrees when $n' = 2$.

3.4.2.3 Alternative segmentation methods

Whilst developing the PTS algorithm we considered alternatives to parts of the segmenting algorithm. This section gives a brief description of alternatives.

The goal of the PTS algorithm is to be an algorithm that can be used when given the most basic information, which, in this case, is a tree topology represented in valid *NEWICK* format [51]. Nothing else is needed to decide where segmentations are made. That means that there does not need to be any other meta-data in order to perform segmentation, which eliminates any extra preprocessing. Allowing the use of meta-data opens the opportunity to use algorithms designed to take advantage of the extra data, which, therefore, make such algorithms case specific. This does not fulfil our goal of creating an algorithm that can be used with the most basic information at hand.

To choose the correct segmentation points in the tree topology, the PTS algorithm does a post-order traversal and counts the desired number of nodes, as we covered in the previous sections. It is also possible to avoid traversing the tree if the *level of a node* is used as the criteria for a segmentation algorithm to investigate. This is possible, since any potential root node is positioned at a specific level in the tree, and if a level is identified, it is possible to calculate the number of nodes in this subtree given the level and the height of the original topology. However, avoiding the traversal is only possible if the tree is strictly bifurcating, because if the tree is multifurcating, any node might have more than two children, which can only be deterministically confirmed by traversing down the tree and counting the actual number of nodes.

Another alternative to traversing the tree topology (without using meta-data) is to pick a set of random nodes. These random segmentation points will then be root nodes of the new subtrees, and the subtrees will be chosen randomly. Then, the optimal subtree size will be obtained by repeatedly selecting random sets of root nodes, testing the total run times. The set which produces the minimum run time will be selected from all the tested sets, after a predefined number of randomly selected sets. The problem when using this approach is that finding the optimal subtree size is non-deterministic, since we can set a limit on how many times a random set of nodes can be chosen, but even with a large number of random sets, there is no insurance that the optimal size will be found. Any optimisation algorithm that has a random element would have the same consequences, for example, *genetic algorithms*, or *hill-climbing algorithms* [52].

3.4.3 Process Management

Once the subtrees have been obtained they can be distributed as tasks (Section 2.3.1) to multiple processes to be computed concurrently. This is done by the *parallel process manager*.

For parallelisation to be justified, it must result in a significant gain in computation time, since it adds extra overhead. The amount of overhead depends on many factors including communication between processes, and preprocessing needed for data decomposition and task decomposition. It is important to keep both of these overheads to a minimum. With the PTS algorithm, our goal is to keep the memory footprints of subtrees as small as possible to reduce the run times of the likelihood calculation. However, it might be the case, depending on hardware, that creating too many subtrees would create too many parallel jobs, which, in turn, might result in suboptimal communication overhead. Therefore, the parallel process manager must be designed to reduce the communication overhead and prevent any deadlocks from occurring.

We shall refer to the computation done on the subtrees as *tasks*, and use the notation Δ_i to denote the i th task relating to the subtree κ_i . Dependencies introduce the problem that a task Δ_i that depends on a task Δ_j cannot be processed normally without the values of the relevant substitution vector being known. We managed the dependency problem by building *priority queues*, which will only allow tasks that do not have any unsolved dependencies to be processed. Priority queues are covered in Section 3.4.3.2. Special care needs to be taken when selecting the scheduling algorithm, since processes will become blocked when waiting for dependencies, as we shall see in Section 3.4.3.1.

The PTS algorithm becomes a *Master/Worker* task decomposition and distribution algorithm [12] (also referred to as the *Master/Slave* algorithm), when combined with any of these two process management algorithms. The *master* instance and *worker* instances are logically separate instances of the program, which run independently as different processes. The master instance does all the initialisation including preprocessing of data and task decomposition. The worker instances are then assigned tasks by the master node, and once they finish their tasks, it is possible that they return some solution to the master instance. In our case, the master instance takes the input and performs the PTS algorithm. It then distributes the tasks to the worker instances, which concurrently perform likelihood computations. How the tasks are distributed, and indeed the definition of the tasks, depends on which one of the two proposed methods are used. This will be discussed in the following sections.

3.4.3.1 Process blocking and waiting time

Processes can become *blocked* in the *priority queue* method. A process p_a is blocked (in a waiting state), if the task Δ_i is the current task assigned to p_a and p_a is waiting for process p_b to finish its work on task Δ_j . Potential computation time is lost if p_a is in a blocked state. We shall call this lost computation time the *waiting time*, and denote it as the function $w(\Delta)$. To identify the process to which a task is assigned, we shall use the notation $\Delta_i^{p_a}$,

where the task Δ_i is assigned to process p_a . The waiting time of $\Delta_j^{p_a}$ can then be calculated by

$$w(\Delta_j^{p_a}) = \alpha(\Delta_j^{p_a}) - \Omega(\Delta_i^{p_a}), \quad (3.4.1)$$

where the function $\alpha(\Delta)$ defines the time at which a task is started, the function $\Omega(\Delta)$ defines the time at which a task has been completed and $\Delta_i^{p_a}$ is the task that p_a processed before $\Delta_j^{p_a}$. The second term $\Omega(\Delta)$ is set to zero if $\Delta_j^{p_a}$ is the first task p_a must process, since all processes start at time zero.

$w(\Delta_j^{p_a})$ can be minimised in several ways:

- *Queue optimisation.* Tasks are ordered in some fashion, and then placed in the primary queue. Processes are then assigned tasks from this queue (FIFO). This is done by the priority queue method, Section 3.4.3.2.
- *Optimising the size of the tasks.* The size of the tasks influence the amount of run time needed to complete them. Smaller tasks will, thus, allow a process to complete the task faster, and that can lead to less waiting time. Caution should be taken that the tasks are not too small, since that may result in excessive communication overhead. We attempt to optimise the size of the tasks when the optimal threshold is being searched for, Section 3.4.1. This optimisation should be done regardless of any of the other optimisation (minimisation) techniques used.
- *Segmenting the dependency $\Delta_j^{p_b}$ which is blocked by and waiting for $\Delta_i^{p_a}$.* The task $\Delta_j^{p_b}$ can be segmented so that the tasks produced by the segmentation can be distributed in order to complete computation of $\Delta_j^{p_b}$ faster. This is not an optimal solution for multiple reasons. Firstly, $\Delta_j^{p_b}$ could already be the smallest possible size, in which case it not be reduced any further. Secondly, p_a will need to interrupt p_b , if $\Delta_j^{p_b}$ is already being processed when p_b becomes blocked, and therefore, the computation being done on $\Delta_j^{p_b}$ will be interrupted. Thus, extra processing and communication overhead is needed.
- *Segmenting the task $\Delta_i^{p_a}$ of the blocked process p_a which is waiting for a dependency $\Delta_j^{p_b}$ to finish.* In many cases, $\Delta_i^{p_a}$ would have at least one subtree where some parts of the subtree is not bound to $\Delta_j^{p_b}$ or any other task. Therefore, p_a can perform the computation on these unbound parts whilst waiting for p_b to finish the work on $\Delta_j^{p_b}$. This method of minimising the waiting time is used as a heuristic in the priority queue method, and is discussed in Section 3.4.5.

3.4.3.2 Priority queues

The scheduling mechanism we use to deal with the dependencies makes use of priority queues assigned to the processes. The PTS algorithm will segment the

tree topology and the resulting jobs are put in a priority queue in the master process. This queue is then sorted, after which the master process distributes the jobs to the worker processes, which store the jobs in their own priority queues. The jobs are stored in some order that minimises the waiting time incurred by processes being blocked. The order in which the jobs are sorted is dictated by the priority assigned to jobs. The nature of the priority depends on which attributes are considered important. Our proposed priority formula is covered in Section 3.4.3.3. For now we shall assume some arbitrary formula.

By providing an example, we shall explain how the priority queues are used. The simulator was run on a randomly generated phylogeny with the following parameters: $n = 20$, $m = 3000$, $p = 5$, $n' = 3$. Here, n is the number of sequences in the sequence alignment, m is the number of sites in the sequence alignment, p is the number of processes and n' is the threshold. The tree topology that was generated and the subtrees that were generated by the PTS algorithm can be seen in Figure 3.13. PTS chose the following segmentation points (root nodes):

$$9, 5, 4, 19, 17, 30, 28, 2, 0.$$

The resulting jobs, one for each subtree, were *sorted by priority*. This produced the ordered priority queue in the master process:

$$\Delta_{19}, \Delta_9, \Delta_{30}, \Delta_5, \Delta_{17}, \Delta_4, \Delta_{28}, \Delta_2, \Delta_0.$$

After prioritising, the master process allocates jobs to processes using a *round-robin scheduling scheme* [37]. This scheduling scheme simply iterates over the list of processes, and at each iteration takes one element from the front of the queue to assign to the current process. The iteration over processes continues until all the jobs have been allocated. In this example, the round-robin scheduler will start by assigning Δ_{19} to p_0 (which stores it in its queue q_0). The iteration continues to p_2 , which is assigned Δ_9 , then p_3 is assigned Δ_{30} , and so forth. Eventually, the round-robin scheme produces the queues shown in Figure 3.14, where the five processes with their respective tasks are shown. The boxes are the duration of the computation done on each job by the process, and the dashed lines are the duration of time that the process is idle. The progression of run time is depicted from top to bottom, where time 0.0 is the time at which the actual concurrent processing of the likelihood computation starts, and 0.00086 is the time at which the likelihood computation is finished (when Δ_0 finishes). Note that the time is measured in seconds.

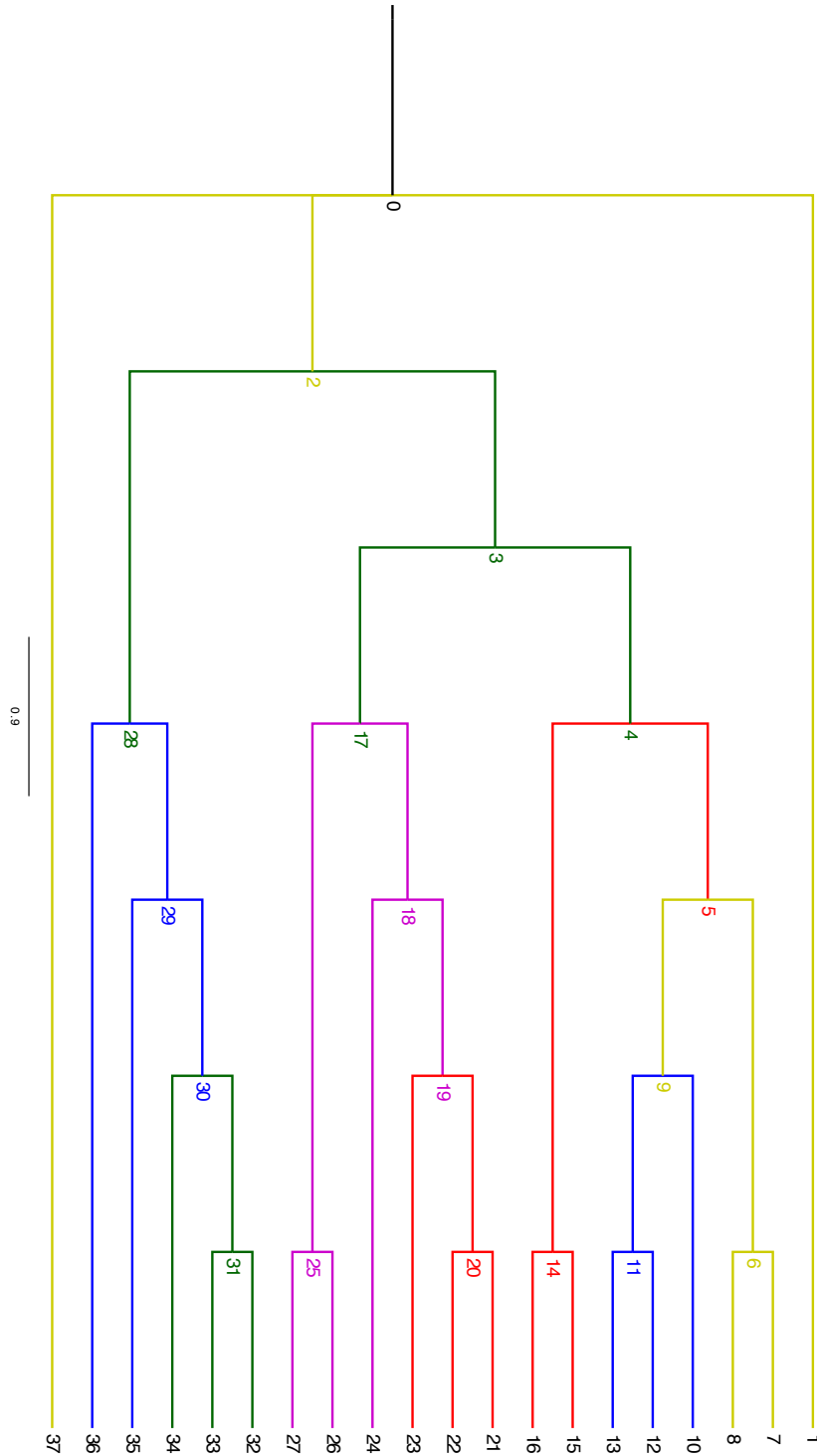


Figure 3.13: A visualisation of how the PTS algorithm segmented the tree and how the tasks (subtrees) were assigned to the process queues. The tasks are colour coded relative to the process priority queues they were assigned to. Red was assigned to q_0 , blue to q_1 , green to q_2 , orange to q_3 and purple to q_4 .

Looking at Figure 3.13, it is clear that Δ_9 , Δ_{19} and Δ_{30} should be of the highest priority, since they have no dependencies and can, therefore, be processed immediately. This is mirrored in the ordering of the priority queues, which allows p_1 , p_2 and p_3 to start computation at time 0.0. Unfortunately, p_3 and p_4 will be idle until Δ_9 and Δ_{19} have been completed respectively, because Δ_5 depends on Δ_9 and Δ_{17} depends on Δ_{19} . The rest of the tasks all have dependencies, and thus they are at the end of the priority queues. The figure shows that Δ_{17} has a longer run time than the other jobs, which all have the same amount of run time. This is due to the subtree of Δ_{17} having a size of $n = 4$, where all the other subtrees have size of $n = 3$.

The simulator estimated that the original run time (that is the run time of the likelihood calculation without any segmentation) would be 0.00678, meaning that there was a estimated speedup (using Equation (2.3.1)) of

$$\begin{aligned} S(p) &= \frac{T(1)}{T(p)} \\ S(5) &= \frac{0.00678}{0.00086} \\ &= 7.82853. \end{aligned} \tag{3.4.2}$$

This dramatic speedup is achieved by computing parts of the likelihood concurrently. However, Figure 3.14 shows how much time the processes spend in an idle state. To improve on this we developed the blocked subtree segmentation heuristic, covered in Section 3.4.5.

3.4.3.3 Priority counter

The priority counter is a real value assigned to a task to define how crucial its computation is; priority rises as the value of the counter increases. The value of the priority counter is calculated by a function that can be dictated by any attributes of the system we deem important.

For our algorithm to work, it is necessary that no deadlocks occur during the concurrent computation of the likelihood calculation, and that jobs should not wait longer on dependencies to finish computation than necessary. The latter requirement is, however, not a necessity for the algorithm to reach a halting state, and is, thus, less important than the former.

Deadlocks can be avoided by forcing tasks that have dependencies to be computed after the tasks that they are dependent upon. This can be accomplished by prioritising tasks in descending level value l , where $l \in \mathbb{N}$. Also, prioritising tasks with fewer dependencies can reduce waiting time. Therefore, we consider two criteria: the level at which the subtree of a task is situated, and the number of dependencies in this subtree. We treat the level as more important and dependencies only to decide priority when two tasks are tied.

This has the effect that a job Δ_i^p with level l_x will always have a position in front of job Δ_j^p with level l_y in the queue of process p , iff $l_x > l_y$. If Δ_i^p has

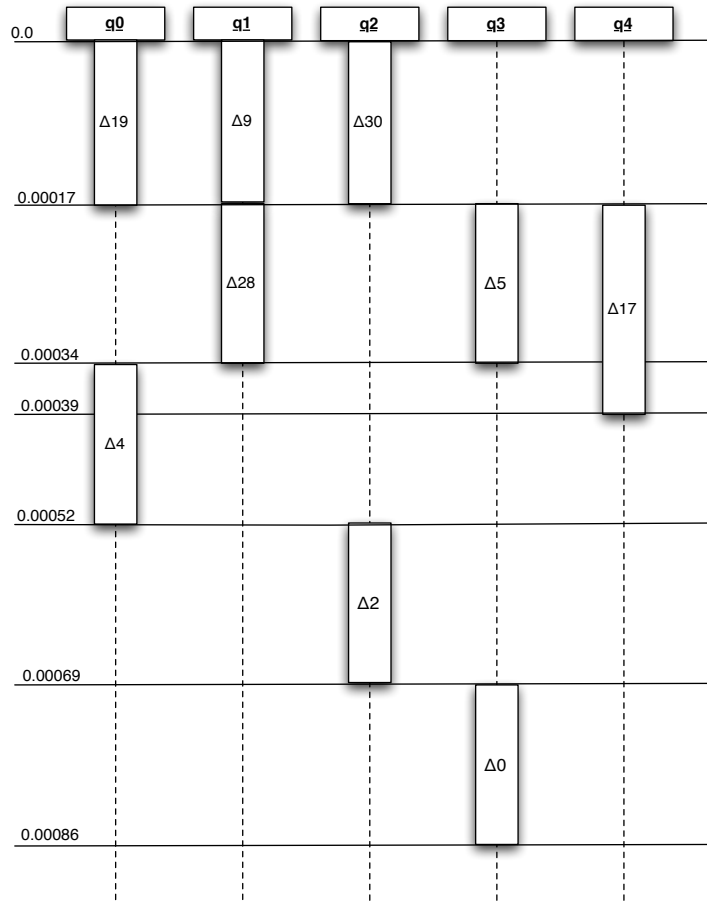


Figure 3.14: An illustration of how the tasks are distributed over the five processes, stored in the queue of each process. The tasks are shown as boxes and the idle time as dashed lines.

d_a dependencies and Δ_j^p has d_b dependencies then Δ_i^p will always be ahead of Δ_j^p in the queue of p , iff $l_x \geq l_y$ and $d_a < d_b$. So, subtrees higher up the tree (closer to the leaf nodes) with fewer dependencies, have priority over subtrees lower down the tree (closer to the root node) with more dependencies.

To calculate the priority we propose Equation (3.4.3),

$$\varphi(l, d) = l + \alpha^{-(d+1)} \quad (3.4.3)$$

where l is the level in the original tree at which the subtree of the job in question is found, d is the number of dependencies which the subtree of the job possess, and $\alpha > 1$ is a predefined constant. One is added to d so that the constraint $\varphi(l, d) < l + 1$ is satisfied; otherwise, a task with no dependencies will erroneously be classified the same as a task that is one level higher. In Equation (3.4.3), the first term will always carry more weight than the second, which fulfills our constraints.

To illustrate with an example, we take the jobs generated for the subtrees in Figure 3.13. Here, Δ_9 , Δ_{19} and Δ_{30} are situated at level six, and have no dependencies. Thus, by using Equation (3.4.3) with $\alpha = 1.01$, we calculate that they have a priority of $\varphi(6, 0) = 6.99009$. Δ_5 is situated at level five and has one dependency (Δ_9), thus giving $\varphi(5, 1) = 5.98029$. This is to say that Δ_9 has a higher priority than Δ_5 , and will therefore always be computed before Δ_5 .

3.4.4 Run time estimation of the segmented phylogeny

The run time of the likelihood calculation using the subtrees produced by the PTS algorithm, given the current threshold, can be estimated once the process manager has prioritised the subtrees.

To estimate the run times, we built a *run time database*, introduced in Section 3.3, in which we recorded run times of various phylogeny sizes. Our simulator, then, uses one of these recorded run times as the estimate, if a matching phylogeny size is found for a subtree. However, when an unrecorded phylogeny size is encountered, the system uses linear regression performed on the previous recorded run times to estimate the run time. The run time of any unrecorded phylogeny size (including subtrees) is added to the database to improve the accuracy and performance. Such a run time database should be constructed for each computer system the simulator is used on. For our experiments, covered in Chapter 4, it was not necessary to use regression, since we populated the database with the appropriate data beforehand.

When the simulator is running, the estimated run time for each chosen threshold value is recorded in the run time results list. This list is then analysed to determine the optimal threshold, after all thresholds have been processed.

3.4.5 Blocked subtree segmentation (BSS)

When using the priority queues method, it is possible to compute some parts of subtrees of jobs that are in a blocked state. We shall yet again use the phylogeny and segmentation shown in Figure 3.13 as an example. Also, we shall say subtree κ_i is the subtree in the figure with root node i as is our previously stated convention. The subtree in this figure that has node 5 as a root node, κ_5 , has one dependency and has a size of $n = 3$; so it is possible to perform computation on subtree κ_6 whilst Δ_5 is blocked and waiting for Δ_9 to finish. We developed the *blocked subtree segmentation algorithm (BSS)* to find subtrees like κ_6 , which can then be computed whilst their parent subtree (in this case κ_5) is blocked.

This algorithm is shown in Algorithm (3.2). It is a derivative of the PTS algorithm, in that it uses post-order traversal, and is also a bottom-up dynamic programming algorithm. However, the BSS algorithm does not count the number of nodes. Instead, it checks if leaf nodes of subtrees have dependencies

or not. The procedure works as follows. The subtree is traversed until a leaf node is reached. If the leaf node has a dependency, a boolean value of *true* is returned. This effectively marks a leaf node as having a dependency. If it does not have a dependency, the boolean value *false* is returned. At inner nodes, all the descendants of an inner node are checked to see if its descendants have been marked as having dependencies, line 9. If a descendant has a dependency, and hence blocked, it is ignored; if not, the descendant is put into the new segments list, which contains the nodes that are the segmentation point for the new subtree(s) (line 13). Segmentations are then made at all the immediate descendants of the current node k that do not have dependencies, if k is labelled as having dependencies (lines 16 and 17).

Algorithm 3.2 Blocked subtree segmentation

```

1: function BSS( $k$ )
2:   if  $k$  is a leaf node with dependency then
3:     return TRUE
4:   else if  $k$  is a leaf node then
5:     return FALSE
6:   else
7:     hasDep := FALSE
8:     segments := []
9:     for every  $child$  in  $k.succ$  do
10:      blocked := BSS( $child$ )
11:      hasDep | = blocked
12:      if not hasDep then
13:        segments.add( $child$ )
14:      end if
15:    end for
16:    if hasDep = TRUE then
17:      SegmentSubtree(segments)
18:    end if
19:    return hasDep
20:  end if
21: end function

```

Suppose $\Delta_i^{p_a}$ is blocked and is segmented with the BSS algorithm, resulting in: $\Delta_i^{p_a}$ (the part which possess the old root node), $\Delta_{i+1}^{p_a}$, and $\Delta_{i+2}^{p_a}$. $\Delta_{i+1}^{p_a}$ and $\Delta_{i+2}^{p_a}$ will need their own priorities defined by the priority Equation (3.4.3). If the round-robin scheduler is used again, it is not guaranteed that the waiting time will not increase, since more dependencies are created which caused the original bottleneck. Therefore, the blocked process must process the new subtrees. Thus, the round-robin scheduler is not used, and the new jobs are simply dropped in the same priority queue as their parent job. The new jobs

will be inserted ahead of their parent job, since the priorities of the new jobs will necessarily be higher than that of their parent.

Design-wise, the BSS algorithm is different from the PTS algorithm. The PTS algorithm is used to roughly segment the whole tree; the BSS algorithm is used to segment a subtree that has dependencies. So, we divide subtree segmentation algorithms into two categories: general segmentation algorithms, and heuristic segmentation algorithms (they can also be called ad-hoc or ‘fine-grain’ segmentation algorithms).

Now, we move back to the example that we introduced in Figure 3.13. Some new subtrees will be created if we run the BSS algorithm on the subtrees already created by the PTS algorithm, but since they are assigned to the same priority queues as their parent subtrees, the colour coding in the figure does not change. The times at which processes are computing and at which they are idle change quite dramatically, as can be seen when comparing Figure 3.14 and Figure 3.15. The latter is the estimation produced by the simulator after the BSS algorithm was run. The first difference that we see here is that p_3 and p_4 , running Δ_5 and Δ_{17} respectively, does not need to wait for their dependencies, they can start computation immediately. They are blocked again once the nodes are reached which have descendants, as can be seen at time 0.00012. Once the dependencies are completed, they can be started again, as can be seen at time 0.00017. Δ_4 is an interesting case, in which p_0 is blocked by p_3 , which is busy with Δ_5 (the dependency of Δ_4). In this case the job can be segmented for computation to carry on. However, p_0 is not blocked once its newly segmented part is done, since by that time (0.00029), p_3 is already done with Δ_5 ; so p_0 just continues the computation on Δ_4 normally.

The complete process is finished in less time, since some of the idle time was eliminated by the extra segmentation done by the BSS algorithm. This time the likelihood calculation was completed in 0.00069 seconds instead of 0.00086 seconds (when only PTS was used). This is, then, an estimated speedup of

$$\begin{aligned} S(p) &= \frac{T(1)}{T(p)} \\ S(5) &= \frac{0.00086}{0.00069} \\ &= 1.246377, \end{aligned} \tag{3.4.4}$$

which is a 1.99755 increase in estimated speedup factor giving a 25.51% increase.

3.5 Summary

In this chapter, we covered the main arguments of this thesis and our proposed parallelisation system.

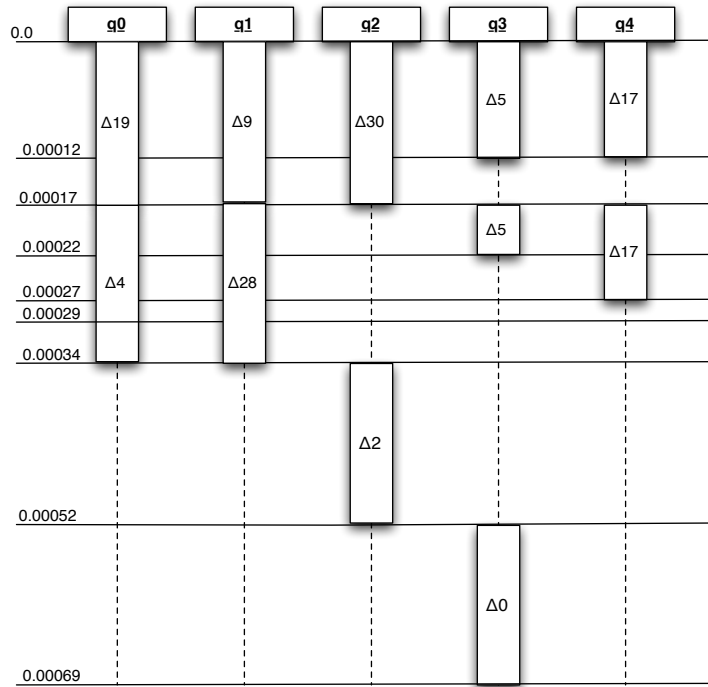


Figure 3.15: The priority queues used with the BSS heuristic.

We started off by covering the complexity analyses of Felsenstein’s algorithm in Section 3.1. In Section 3.1.1, we looked at calculating the theoretical run time of the algorithm. Thereafter, in Section 3.1.2, we looked at calculating the theoretical memory footprint of the data structures needed to perform the likelihood calculation. Then, we covered the effects of the memory footprint on the likelihood algorithm run times in Section 3.2. The effects of the two data decomposition methods, decomposition over sites and decomposition the phylogeny, were discussed in Section 3.2.1. Here we saw that the memory footprint of data decomposition over phylogeny (our proposed method) is dramatically smaller than the memory footprint of data decomposition over sites. In Section 3.2.2, we looked at effects of memory footprints on cache and the likelihood calculation run times. We concluded that when nucleotide models are used, small trees can fit in the cache. As a result, less cache swapping occurred, which, in turn, results in faster run times than expected. The gradient of the line plotted on the run time data quickly increased as the tree sizes increased, due to the increased cache swapping. For codon models, we found that even the smallest trees cannot fit into cache, and as a result, the gradient increases almost linearly. Then, in Section 3.3, we covered how we estimated the run times by using recorded run times or linear regression on recorded run times if a likelihood calculation had not been computed for a given phylogeny size, and how a database of run times was built for use in the system

that implements our parallelisation method. In Section 3.4, we cover our proposed system that finds the optimal subtree size during preprocessing, which allows the likelihood calculation to be performed in less time. The first step is when the threshold is chosen, which is used to decide segmentation options in the phylogeny 3.4.1. The phylogenetic tree segmentation algorithm (PTS) was covered in Section 3.4.2. The topic discussed in the subsections were: segmenting the tree in Section 3.4.2.1, the influence of subtree dependencies introduced by the segmentation process and the substitution vectors needed to pass the results from one subtree to a dependence, which was covered in Section 3.4.2.2, and alternatives to the methods we propose in Section 3.4.2.3.

How parallel processing is effected and managed was covered in Section 3.4.3. Here, we cover how processes are blocked and the waiting time that is incurred, discussed in Section 3.4.3.1. We also proposed solutions to this problem. Our preferred solution, priority queues, and how they work was covered in Section 3.4.3.2. The function calculating the priority is discussed, thereafter, in Section 3.4.3.3.

Lastly, in Section 3.4.5, we suggested a heuristic named the blocked subtree segmentation (BSS) algorithm that extends the PTS algorithm.

Chapter 4

Results

In this chapter we discuss different tests we have devised to measure the performance of the segmentation algorithm on a variety of phylogenies, and the results that were obtained.

Section 4.1 describes the data sets that were used in the experiments. Then, in Section 4.2, the run time estimation experiments are described, the results are presented and then discussed.

4.1 Data sets

To test our algorithm we used sequence alignments of the *influenza A H1N1 virus* [53], the *human immunodeficiency virus (HIV)* (specifically HIV-1) [54] and randomly generated data.

The H1N1 and HIV-1 sequence alignments were chosen for two reasons. Firstly, the large number of unique sequences available for every gene enabled us to build large realistic phylogenies. Secondly, we stated previously in Section 3.4.2.2 that our algorithm will perform better on trees that are balanced like and poorer on ladder-like trees, since waiting times present a larger problem when a phylogeny is more ladder-like. Therefore, it was essential that we ran the algorithm on both types of trees.

H1N1 sequence alignments tend to produce ladder-like phylogenies, whereas HIV-1 sequence alignments tend to produce more balanced phylogenies. Ladder-like trees usually indicate *adaptation*, such as *immune escape*, which in HIV-1 typically occurs intra-host, but not inter-host [55]. However, with H1N1 (and other influenza viruses) we typically see such adaptation in both intra-host and inter-host data [56]. The sequence alignments we present are made up of samples taken from different hosts (i.e., inter-host data), therefore we expect to see H1N1 to give ladder-like trees and HIV-1 to give balanced like trees.

Our phylogenies were constructed using *Phyml* [22] on the *Moby* servers at the *Institut Pasteur* [57, 58]. The resulting phylogenies for the H1N1 NP, H1N1 HA, HIV-1 POL and HIV-1 ENV genes are shown in Figures 4.1 to 4.4,

which have been rendered using the FigTree software package [59]. It is clear that the trees of the H1N1 genes are ladder-like trees and that the trees of the HIV-1 genes are relatively balanced trees.

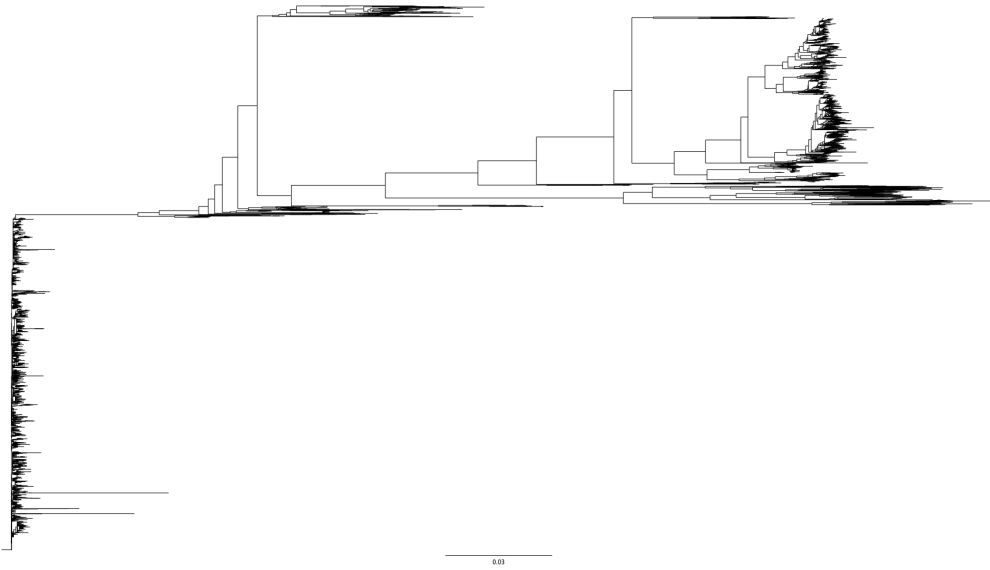


Figure 4.1: Phylogeny constructed for the H1N1 HA gene sequence alignment from the NCBI influenza database.

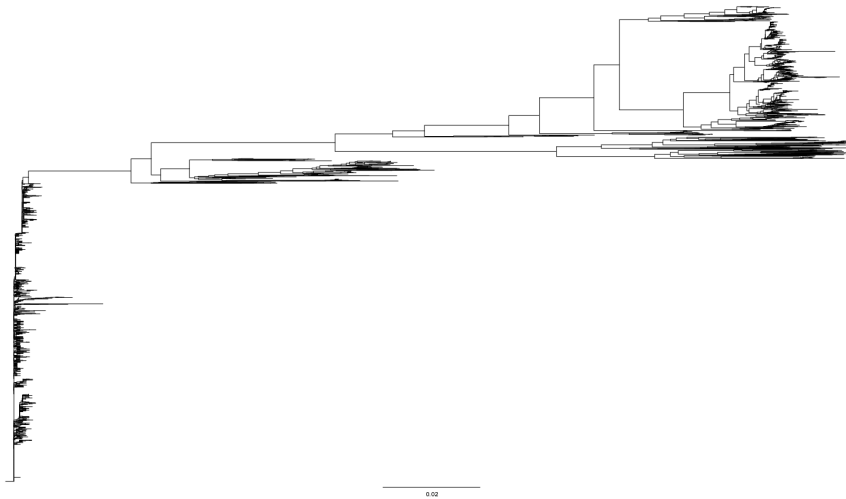


Figure 4.2: Phylogeny constructed for the H1N1 NP gene sequence alignment from the NCBI influenza database.



Figure 4.3: Phylogeny constructed for the HIV-1 ENV gene sequence alignment from the Los Alamos HIV database.

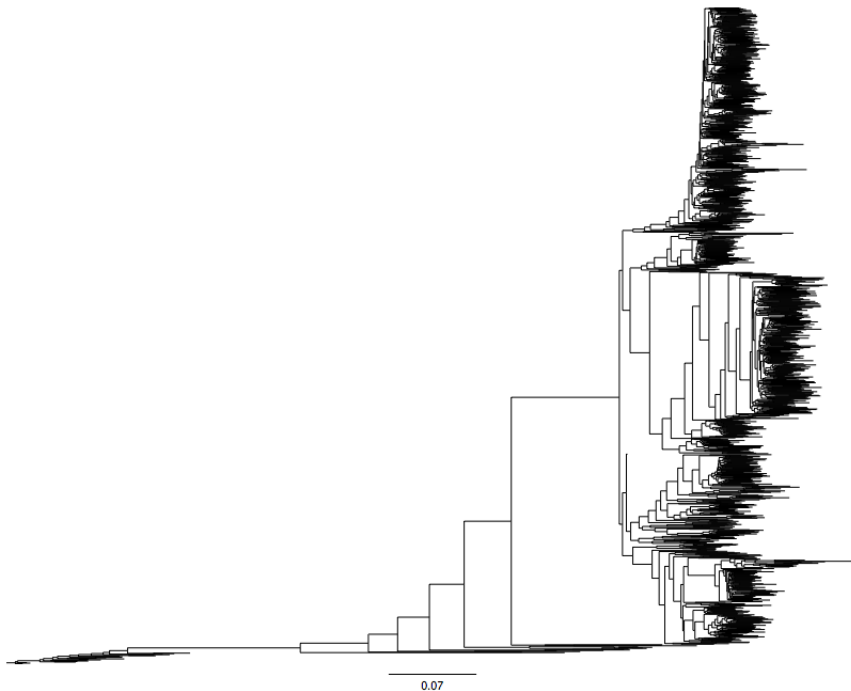


Figure 4.4: Phylogeny constructed for the HIV-1 POL gene sequence alignment from the Los Alamos HIV database.

In the above figures we can clearly see very densely grouped branches. This is due to the large sizes of these sequence alignments. The H1N1 HA gene data set consists of 4928 DNA sequences with 1922 sites, whereas the H1N1 NP gene

data set has 3405 DNA sequences with 1666 sites. For HIV-1 POL there are 1250 DNA sequences with 3363 sites and for HIV-1 ENV there are 1963 DNA sequences with 3352 sites. The results obtained when using these sequence alignments are discussed in the following section.

4.2 Speedup as a function of phylogeny size

Our simulator was used to develop the theory behind our proposed parallelisation system and to test and verify it throughout development. We therefore decided to employ the simulator for the evaluation of our proposed system. This is apt, since any software that intends to implement our proposed work will use the results (estimates) of such a simulator to pick an optimal segmentation scheme.

In Section 3.2.2 we showed how various phylogeny sizes performed when using HyPhy on a cluster. These, and other estimates, are used by our simulator to estimate run times for segmentation schemes, as covered in Section 3.4.4. We first compared run time estimations of various phylogeny sizes for both nucleotide and codon models in Section 4.2.1. We then compared the run time estimations of genes chosen from the H1N1 and HIV-1 datasets over various numbers of processes in Section 4.2.2. Throughout this chapter, we assume that all processes are run concurrently; that is, each process is run on a separate processor.

4.2.1 Speedup as a function of the number of processes

In these experiments we measured the estimated run time speedup of the likelihood calculation when using our proposed system. Similar to Section 3.2.2, three experiments were conducted for both model types, each experiment having a constant number of sites: 300, 1200, and 3000. Sequence alignments of 10, 100, 500 and 1000 sequences were used for each experiment. Each DNA string in the sequence alignments were generated by selecting a random nucleotide uniformly from the set of nucleotides, after which HyPhy was used to infer phylogenies for them; this was done one hundred times per phylogeny size to obtain a verity of topologies to test. We repeated the experiments with 1, 5 and 10 processes. As before we shall denote the number of sequences as n , the number of sites as m , the number of processes as p and the size of the alphabet as N ($N = 4$ for nucleotide models and $N = 61$ for codon models).

Figures 4.5a to 4.5f show the mean run time speedup obtained when using our proposed system for each phylogeny size in these experiments. The left hand column shows the experiments where nucleotide models were used, whereas the experiments shown in the right hand column used codon models. Note that, the definition of speedup (Equation (2.3.1)) differs for the two cases where $p = 1$. In this case, the speedup was measured by comparing the original

run time $T(1)$ and the run time obtained when segmentation was performed using only one process $T(1)'$, which gives

$$S(1) = \frac{T(1)}{T(1)'} \quad (4.2.1)$$

As a summary of two of these experiments, we provide Tables 4.1 to 4.6 to illustrate the correlation between the increase in speedup and the decrease in subtree sizes, observed in all the experiments. The tables show the number of subtrees created by the PTS algorithm, the threshold n' that was chosen as optimal, the speedup $S(1)$ and the size of the memory footprint of the original phylogeny (Equation (3.1.11)) to number of subtrees ratio (i.e., the average size of a subtree). The results in Tables 4.1 to 4.3 are for the cases where nucleotide models are used for processing sequence alignments with 300, 1200 and 3000 sites on only one process, whereas the results in Tables 4.4 to 4.6 are for cases where codon models are used for processing sequence alignments with 300, 1200 and 3000 sites on ten processes.

$m = 300, p = 1$				
n	#subtrees	n'	$S(1)$	$\frac{\phi(N,m,n)}{\#subtrees}$ (in bytes)
10	1	10	0	5993.0 B
100	17	6	1.08	3592.529 B
500	86	6	1.08	3556.662 B
1000	171	6	1.08	3578.204 B

Table 4.1: Performance metrics for the experiments where $N = 4$, $m = 300$ and $p = 1$.

$m = 1200, p = 1$				
n	#subtrees	n'	$S(1)$	$\frac{\phi(N,m,n)}{\#subtrees}$ (in bytes)
10	2	5	1.31	7496.5 B
100	27	4	3.04	5595.296 B
500	147	4	3.194	5141.993 B
1000	297	4	3.174	5090.481 B

Table 4.2: Performance metrics for the experiments where $N = 4$, $m = 1200$ and $p = 1$.

Looking at Figures 4.5a to 4.5f, we saw an increase in run time speedup, as the number of sequences increase, up to a point where a plateau is reached for every case where nucleotide models were used. This initial speedup was obtained due to the PTS algorithm segmenting the original tree into smaller

$m = 3000, p = 1$				
n	#subtrees	n'	$S(1)$	$\frac{\phi(N,m,n)}{\#subtrees}$ (in bytes)
10	3	4	2.34	10997.666 B
100	31	4	6.32	10679.774 B
500	150	4	6.33	11039.153 B
1000	296	4	6.29	11188.760 B

Table 4.3: Performance metrics for the experiments where $N = 4$, $m = 3000$ and $p = 1$.

$m = 300, p = 10$				
n	#subtrees	n'	$S(10)$	$\frac{\phi(N,m,n)}{\#subtrees}$ (in bytes)
10	1	10	1	5993.0 B
100	27	4	5.61	2261.962 B
500	109	5	9.22	2806.174 B
1000	215	5	9.35	2845.920 B

Table 4.4: Performance metrics for the experiments where $N = 4$, $m = 300$ and $p = 10$.

$m = 1200, p = 10$				
n	#subtrees	n'	$S(10)$	$\frac{\phi(N,m,n)}{\#subtrees}$ (in bytes)
10	8	1	1.45	749.125 B
100	30	4	16.69	2035.766 B
500	148	4	25.38	2066.709 B
1000	290	4	26.90	2109.906 B

Table 4.5: Performance metrics for the experiments where $N = 4$, $m = 1200$ and $p = 10$.

subtrees, which could then be computed concurrently on additional processors for $p > 1$; as well as be computed faster than the original in rapid succession, since there was less cache swapping as described in Section 3.2.1. However, a plateau was reached at the point where the speedup started to remain relatively constant.

To explain the plateau we look at the metrics shown in Tables 4.1 to 4.6. We notice that the number of subtrees produced by the PTS algorithm increase linearly as the size of the original phylogeny increases. However, the subtree size decrease and then becomes fairly constant (shown in the last columns of Tables 4.1 to 4.6). Therefore, we observe the plateau, since the time gained from smaller subtrees stops contributing to the speedup, and the number of subtrees distributed over the same number of process increase.

$m = 3000, p = 10$				
n	#subtrees	n'	$S(10)$	$\frac{\phi(N,m,n)}{\#subtrees}$ (in bytes)
10	8	1	2.89	749.125 B
100	44	3	32.69	1388.022 B
500	232	3	53.46	1318.418 B
1000	451	3	57.19	1356.702 B

Table 4.6: Performance metrics for the experiments where $N = 4$, $m = 3000$ and $p = 10$.

A higher speedup was measured for longer sequences, since these phylogenies have larger memory footprints, which benefited more from being segmented into subtrees than phylogenies with smaller memory footprints. As expected, the speedup also increases as p is increased, shown in Figures 4.5a, 4.5c and 4.5e.

Interestingly, super-linear speedup was still obtained when $p = 1$, shown in Figure 4.5a, even though no parallel processing was done. This is simply an effect of the improved run times due to cache swapping, as was discussed in Section 3.2.2. One can clearly see that the sum of the run times of smaller subtrees are less than the run time of the larger original tree they belong to. This can be verified using the results shown in Figure 3.5.

Figure 4.5b shows the run time speedups for $p = 1$ where codon models are used. Here we have a different situation from what was seen previously with nucleotide models. For both $m = 300$ and $m = 1200$ no speedup increase was observed, but for $m = 3000$ super-linear speedup was again observed. This can be explained by the gradients seen in Figure 3.7. As stated previously, the gradients for $m = 300$ and $m = 1200$ are not as steep as that of $m = 3000$. This means that in these two cases the sum of run times for smaller subtrees is greater than the run time of the original tree. Therefore, the simulator will not choose any segmentation scheme, since no scheme produces any speedup, hence, the speedup obtained in these cases is 1. This is not the case when $m = 3000$. Here the sum of run times is again less than that of the original tree. Thus, speedup is observed again. The cases for $p = 5$ and $p = 10$, shown in Figures 4.5d and 4.5f respectively, show the general trend of speedup until a plateau was researched, and the run time speedup that increases as the number of processes increases.

4.2.2 Run time estimation as a function of the number of processes

In these experiments we measured run time estimations of the likelihood calculation over an increasing number of processes. Sequence alignments and their corresponding phylogenies for the H1N1 HA, H1N1 NP, HIV-1 ENV and HIV-

1 POL genes (covered in Section 4.1) were used for these experiments. For each phylogeny we ran an experiment for a number of processes ranging from 1 to 50. The graphs in Figures 4.6 to 4.9 depict the speedup factor gained as the number of processes is increased. Figures 4.6 and 4.7 shows the results for the H1N1 HA and NP data sets, whereas Figure 4.8 and 4.9 shows the results for the HIV-1 ENV and POL data sets. The green plots show the speedup factors recorded when the PTS algorithm was used, and the red plots show the speedup factors recorded when the BSS algorithm was used in addition to the PTS algorithm. Bumps in the estimated run times, especially noticeable in Figures 4.8a and 4.9a, are due to variance in the database inherited from noise in the recorded run times.

Figures 4.10 and 4.11 depict the waiting times observed in the experiments which obtained the optimal run time as chosen by the simulator. In Section 3.4, we described how the simulator finds the optimal threshold, which results in the optimal estimated run time.

In these experiments we once again observed speedups up to a point where the speedup hits a plateau, which is typical of parallel processing as covered in Section 2.3.2. Super-linear speedup was observed for the estimations where nucleotide models were used, and sub-linear speedup was observed for the estimations where codon models were used. We can again point to the run times of the smaller subtrees compared to the larger original phylogeny for the results obtained when using nucleotide models, and the gradients seen in Figure 3.7 for the results obtained when codon models were used.

One noticeable exception is speedup over processes for the H1N1 NP alignment, Figure 4.7a. Although super-linear speedup was obtained, it is only obtained up to roughly $p = 31$. This can be explained by looking at the topologies of the phylogenies and the waiting times. The phylogeny of H1N1 NP, shown in Figure 4.2, is the most ladder-like of the four, and these trees will suffer more from waiting times than balanced trees, because the segmentation points in ladder-like trees will inherently be more likely to be descendants of other segmentation points as Section 3.4.2.2 discussed. This is verified by the results shown in Figures 4.10 and 4.11, which indicates that the waiting times in the H1N1 NP runs were twice as long as those of the H1N1 HA runs (the second longest). Therefore, as predicted, we observe that the speedup obtained for the ladder-like trees of H1N1 is less than that for the balanced trees of HIV-1.

BSS was designed to decrease the time that processes are idle, thus it should reduce waiting times. Table 4.7 shows the mean percentage increase in run time speedups from when the PTS algorithm was used to when the BSS algorithm was also used. As can be seen from this table there is no clear way to predict how much the BSS algorithm will induce an increase in speedup based on the topology of the tree. However, the extra speedup obtained is noteworthy, especially when the long run times we observe for large phylogenies are considered. Thus, we suggest that the BSS algorithm be used in all cases,

seeing that this extra speedup can be obtained by only adding the negligible computation of the BSS algorithm to the standard PTS algorithm.

	Nucleotides	Codons
H1N1 HA	6.73	8.37
H1N1 NP	8.08	2.82
HIV-1 ENV	6.60	0.42
HIV-1 POL	28.87	16.41

Table 4.7: The mean percentage increase in run time speedups from PTS to BSS on the chosen sequence alignments.

4.3 Summary

In this chapter we presented and discussed the results of the experiments we performed to test the PTS and BSS algorithms on synthetic and real data sets. We started by describing the data sets in Section 4.1, giving the dimensions of the sequence alignments, as well as presenting the phylogenies that were inferred for these alignments. Then in Section 4.2 we covered two types of experiments that we performed. Firstly, Section 4.2.1 covered the influence of phylogeny sizes on the run time of likelihood calculation. We saw that in most cases the run time speedup increased as the size of the phylogenies increased up to a point where a plateau was reached. This was mainly due to how the size of phylogenies influences cache and memory swapping as well as the amount of computation that needs to be performed to calculate the likelihood. Secondly, Section 4.2.2 covered the influence of the number of processes used for parallelisation on the likelihood calculation. The same trend in the increase of speedup factor was observed as in the previous type of experiment; however, in this case, the major contributing factor was the distribution of the computation time over different processors. As expected, the speedup factor obtained was greater when nucleotide models were used as opposed to when codon models were used in both types of experiments.

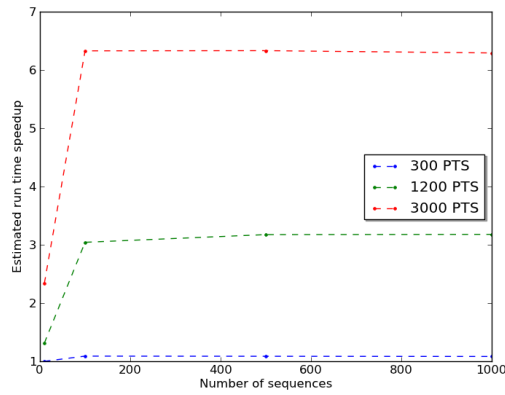
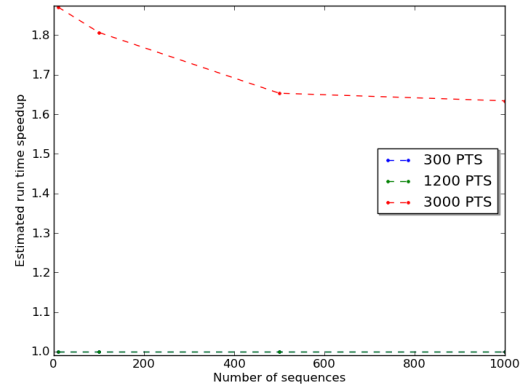
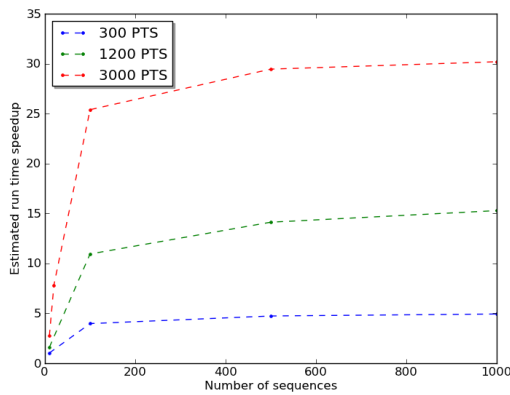
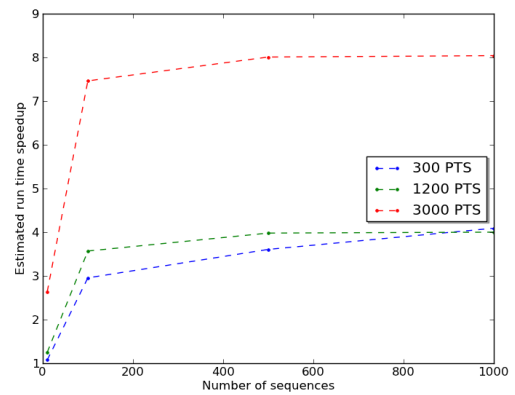
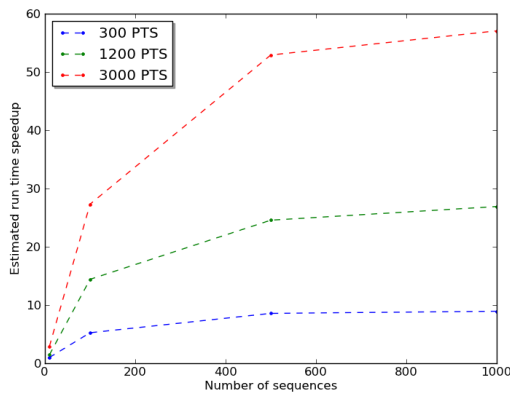
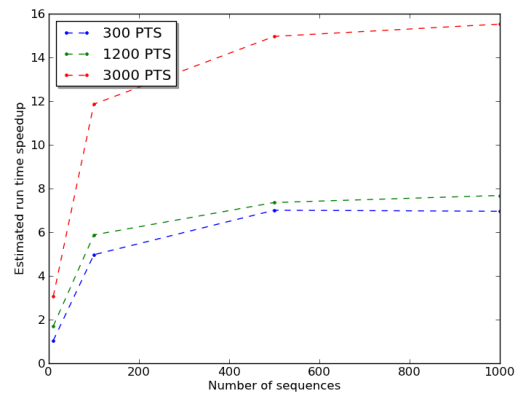
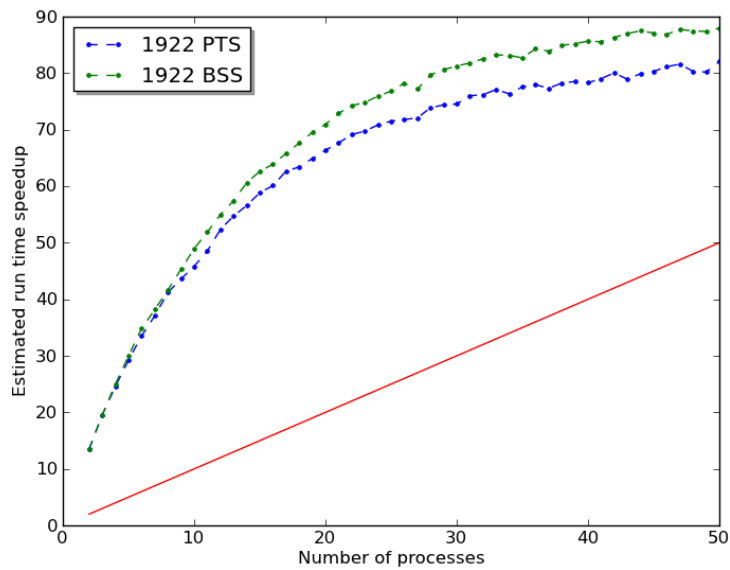
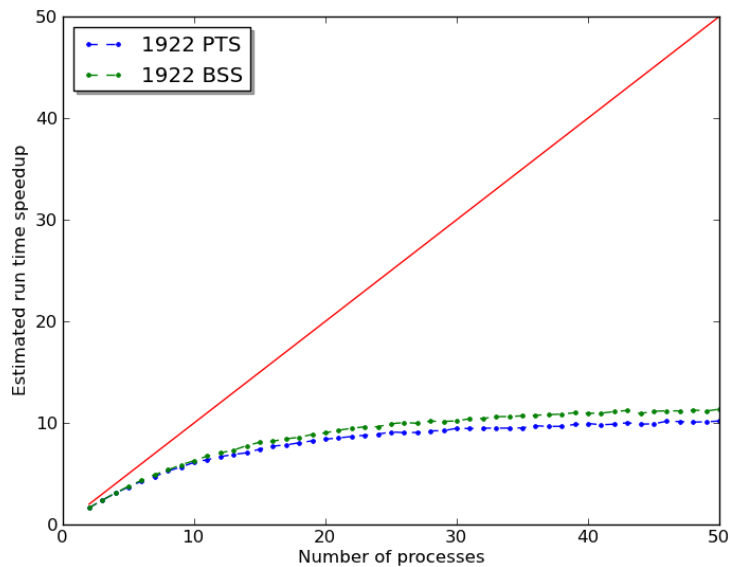
(a) Nucleotide $p = 1$ (b) Codon $p = 1$ (c) Nucleotide $p = 5$ (d) Codon $p = 5$ (e) Nucleotide $p = 10$ (f) Codon $p = 10$

Figure 4.5: Estimated run time speedup as a function of phylogeny size using nucleotide and codon models and for different numbers (p) of processes. Sequence alignments with 10, 100, 500 and 1000 sequences were used. This was done for 300 (blue), 1200 (green) and 3000 (red) sites each. Only the results of the PTS algorithm are shown. Speedup is calculated using our database of run time estimates for single processor likelihood computations on different phylogeny sizes (see Section 3.4.4).

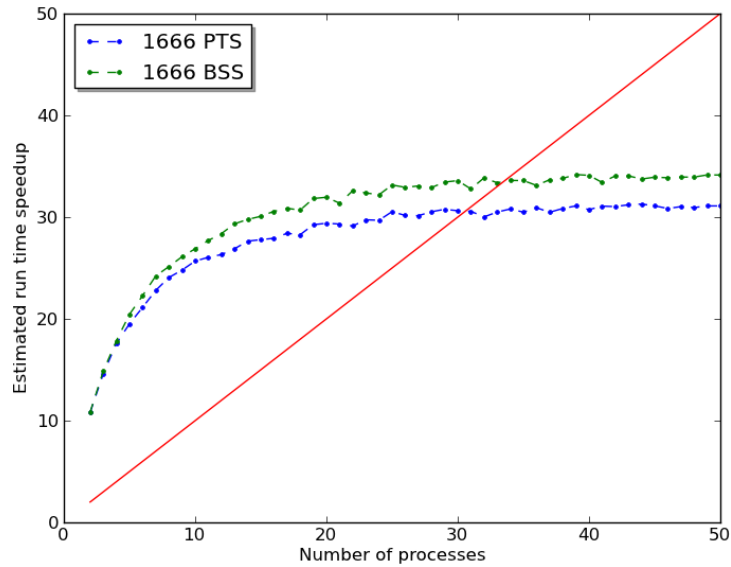


(a) Nucleotide H1N1 HA

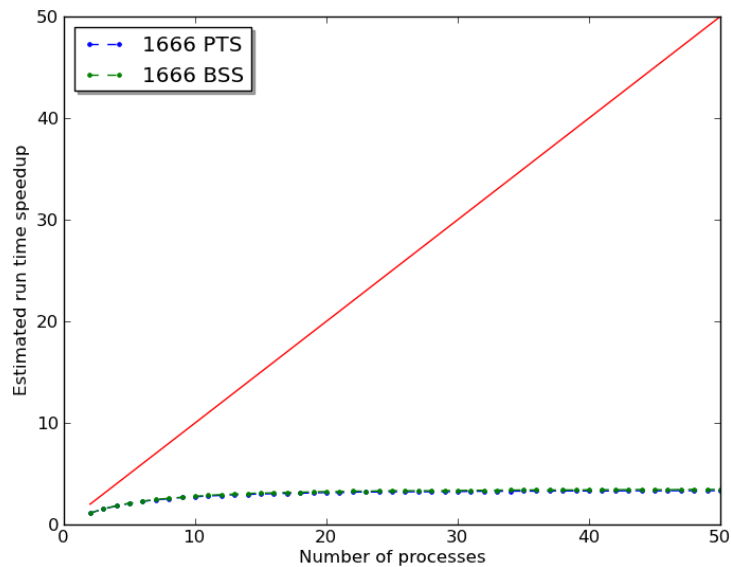


(b) Codon H1N1 HA

Figure 4.6: Estimated run time speedup for the H1N1 HA sequence alignments when using nucleotide and codon models over a number of processes ranging from 1 to 50. The results of the PTS and BSS algorithms are shown in blue and green respectively. Speedup is calculated using our database of run time estimates for single processor likelihood computations on different phylogeny sizes (see Section 3.4.4). The blue lines show the coordinates (x, y) , where $x = y$, which is what the speedup would be if linear speedup was obtained.

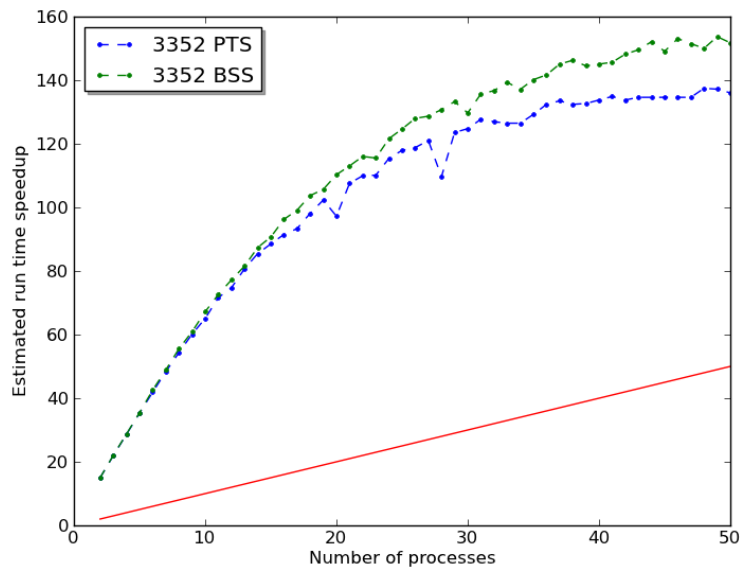


(a) Nucleotide H1N1 NP

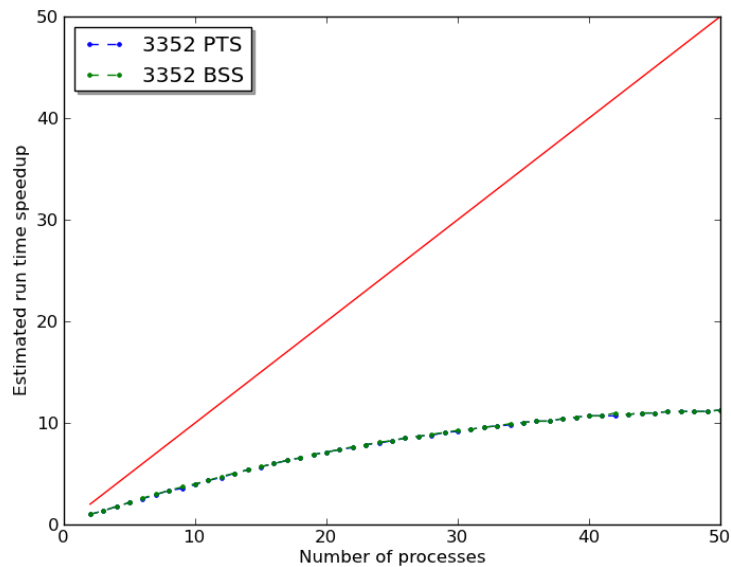


(b) Codon H1N1 NP

Figure 4.7: Estimated run time speedup for the H1N1 NP sequence alignments when using nucleotide and codon models over a number of processes ranging from 1 to 50. The results of the PTS and BSS algorithms are shown in blue and green respectively. Speedup is calculated using our database of run time estimates for single processor likelihood computations on different phylogeny sizes (see Section 3.4.4). The blue lines show the coordinates (x, y) , where $x = y$, which is what the speedup would be if linear speedup was obtained.

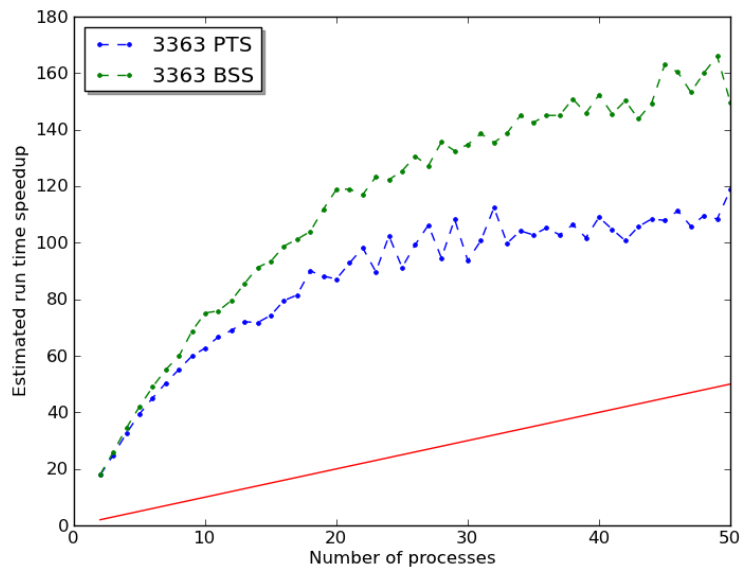


(a) Nucleotide HIV-1 ENV

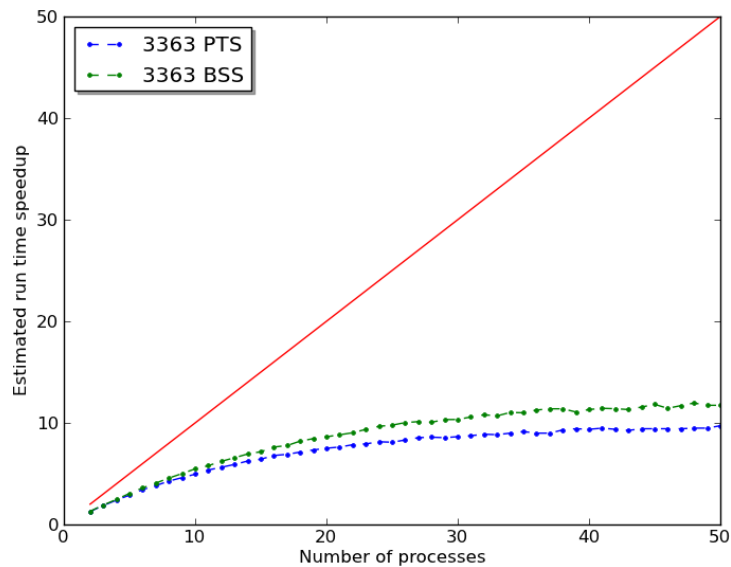


(b) Codon HIV-1 ENV

Figure 4.8: Estimated run time speedup for the HIV-1 ENV sequence alignments when using nucleotide and codon models over a number of processes ranging from 1 to 50. The results of the PTS and BSS algorithms are shown in blue and green respectively. Speedup is calculated using our database of run time estimates for single processor likelihood computations on different phylogeny sizes (see Section 3.4.4). The blue lines show the coordinates (x, y) , where $x = y$, which is what the speedup would be if linear speedup was obtained.



(a) Nucleotide HIV-1 POL



(b) Codon HIV-1 POL

Figure 4.9: Estimated run time speedup for the HIV-1 POL sequence alignments when using nucleotide and codon models over a number of processes ranging from 1 to 50. The results of the PTS and BSS algorithms are shown in blue and green respectively. Speedup is calculated using our database of run time estimates for single processor likelihood computations on different phylogeny sizes (see Section 3.4.4). The blue lines show the coordinates (x, y) , where $x = y$, which is what the speedup would be if linear speedup was obtained.

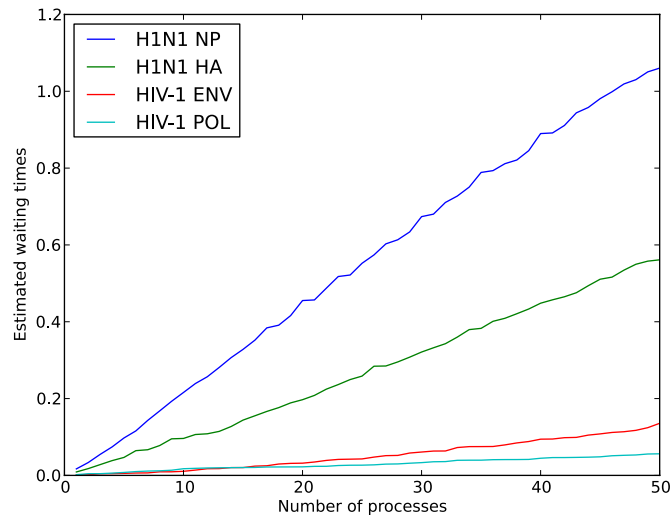


Figure 4.10: Estimated waiting times for the H1N1 NP, H1N1 HA, HIV-1 ENV and HIV-1 POL sequence alignments when nucleotide models are used over a number of processes ranging from 1 to 50.

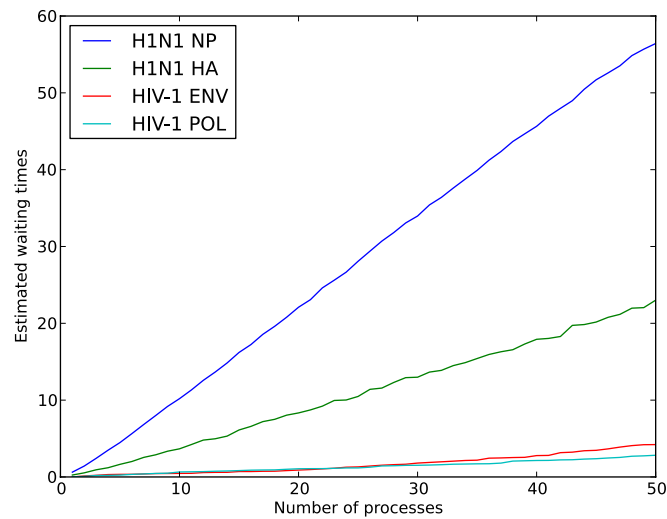


Figure 4.11: Estimated waiting times for the H1N1 NP, H1N1 HA, HIV-1 ENV and HIV-1 POL sequence alignments when codon models are used over a number of processes ranging from 1 to 50.

Chapter 5

Conclusion

In this chapter, we make conclusions about our proposed parallelised likelihood calculation system. In Section 5.1, a summary of our findings is given. Then, in Section 5.2, we present the contributions that were made by the work in this thesis. Lastly, in Section 5.3, we give suggestions for future work that can build on our contributions.

5.1 Summary of findings

In Section 3.1, we defined complexity formulae for both the run time and space complexity of Felsenstein's tree pruning algorithm. With these formulae we showed that the memory footprint of Felsenstein's algorithm would be reduced when data decomposition is performed over sites and over the phylogeny. It was shown in Figures 3.2 through 3.4 that data decomposition over the phylogeny is more effective at reducing the memory footprint than data decomposition over sites, especially when codon models are used.

Then, in Section 3.2.2, we presented the run times of likelihood calculations performed with HyPhy for different phylogeny sizes. For nucleotide models, the results showed that the run times increased drastically once a relatively small phylogeny size is reached (see Figure 3.5). We concluded that this increase was caused by the cache limit being reached, at which point cache swapping slowed the calculation down. For codon models, the results showed a linear increase in run times. Here, we concluded that the memory footprint was always too big to fit in cache, and, therefore, swapping occurred even from the smallest phylogeny sizes. These results were supported by the theoretical limits of the phylogeny sizes that can fit into L1 cache shown in Tables 3.1 and 3.2.

We then introduced our proposed system, which included the PTS and BSS algorithms, the methods of choosing the optimal threshold to use in these algorithms and the various parts of the process management system. Examples of the priority queues were given for the case where the PTS algorithm was used (Figure 3.14), and the case where the BSS algorithm was used (Figure 3.15).

We saw a distinct decrease in waiting times, and, therefore, a decrease in the run time from the run where PTS was used to the run where BSS was used.

We wanted to test our algorithms on both balanced trees and ladder-like trees, since we hypothesised that our algorithms will perform better on the former than the latter due to waiting times that present a larger problem if a phylogeny is more ladder-like. For this purpose we needed sequence alignments that would produce these types of trees. In Section 4.1, it was shown that the H1N1 genes were more ladder-like, whilst the HIV-1 genes were more balanced.

Then, we ran experiments to evaluate the estimated run times when the phylogeny sizes were increased, and when the number of processes was increased. For the former we used synthetic data, and for the latter the H1N1 and HIV-1 sequence alignments were used. This was done for both nucleotide and codon models.

A general trend of an increase in run time up to a point where a plateau was reached was observed in the experiments where we increased the phylogeny sizes. The one exception was the case of codon models being used on only one process. This initial speedup was obtained due to the small subtrees that could be computed faster than the original in rapid succession. For the other cases, the plateau was reached when the number of subtrees that need to be computed increased to a point where the time gained from using smaller subtrees stops contributing to the speedup and the number of subtrees distributed over the same number of process increased linearly.

The influence of distributing the computation of subtrees on the total run times was notable, for both nucleotide and codon models. As expected, the speedup achieved when nucleotide models were used was higher than when codon models were used, which can be explained by the faster than expected run times of smaller trees, and then the rapid increase in run times which is observed as the size of the trees grows.

5.2 Contributions

In reaching the objective of developing parallel likelihood calculations for phylogenetic trees, we made the following contributions:

- Our proposed parallelisation system performs parallelisation on the Felsenstein's pruning algorithm in order to decrease the run time of the likelihood calculation of phylogenies, which also increases the size of phylogenies that we are able to analyse. This is done by performing data decomposition on the phylogeny of interest.

The system can be incorporated into any phylogenetic software package as a preprocessing step to the main computation, using parallel programming frameworks such as threading libraries or message parsing interface (MPI). Heuristics will have to be built into the system for when a phy-

logeny is changed, if the system is incorporated into a package that is used for phylogeny inference. Otherwise, if our system is used in parameter estimation, such heuristics will not be needed, since the tree does not change. Codon modelling is such an application.

Most previous work on parallelisation in phylogenies has focused on phylogeny inference. Thus, our approach opens new possibilities for software that does not focus on phylogeny inference.

- The phylogenetic tree segmentation (PTS) and blocked subtree segmentation (BSS) algorithms can be used independently of our proposed system. Therefore, the segmentation of a phylogeny can be provided in other implementations that do not share the design of our system, by incorporating the PTS and BSS algorithms. For example, another system might use another job distribution strategy, such as MapReduce [60].
- Our study on how memory effects run times shows interesting and important trends in likelihood run times based on phylogeny size. These results can be used, for example, to design a model that predicts likelihood run times based on phylogeny size, which would be useful for obtaining more accurate estimation with the simulator.

5.3 Suggestions for future work

After the analysis and testing of our proposed system, it was concluded that our proposed methodology is promising. There are many possible heuristics that can be applied in order to improve the performance even further. Heuristics to reduce waiting times, other than the blocked subtree segmentation (BSS) algorithm, can be applied to the phylogenetic tree segmentation (PTS) algorithm. Heuristics can also be applied to the parallel process manager, which dictates how queues are managed. For example, it is possible to better sort jobs in the process queues, so that dependent subtrees have a higher possibility of sharing a queue.

Job distribution strategies other than the master/worker approach that we chose can also be used and compared. For example, the MapReduce strategy is a popular option, which is well suited for use with our PTS algorithm.

Another avenue for improvement is to integrate the RAxML GPU approach (covered in Section 2.5.3)[16] with the PTS algorithm in current software. We believe that this integration has the potential to work well, since the calculation of the transition probability matrices are still a bottleneck that can be improved. There are, however, many open questions about how the two approaches will interact that need to be investigated in order to assess the feasibility of such an integration.

The next step is to integrate our system into a phylogenetic software package used in practice. We aim to implement it as part of the HyPhy package,

which was used to produce the empirical results presented here. This would allow us to report actual, rather than estimated run time results. Some heuristics might be needed to effectively run such a modified version of the software, depending on the architecture of the cluster, as suggested in Section 5.2.

List of References

- [1] M. Larkin, G. Blackshields, N. Brown, R. Chenna, P. McGettigan, H. McWilliam, F. Valentin, I. Wallace, A. Wilm, R. Lopez *et al.*, “Clustal W and Clustal X version 2.0,” *Bioinformatics*, vol. 23, no. 21, p. 2947, 2007.
- [2] I. Letunic and P. Bork, “Interactive Tree Of Life (iTOL): an online tool for phylogenetic tree display and annotation,” *Bioinformatics*, vol. 23, no. 1, p. 127, 2006. [Online]. Available: <http://itol.embl.de/>
- [3] NCBI, “Genbank growth statistics,” 2009. [Online]. Available: <http://www.ncbi.nlm.nih.gov/genbank/genbankstats.html>
- [4] —, “Genbank overview,” 2004. [Online]. Available: <http://www.ncbi.nlm.nih.gov/genbank/GenbankOverview.html>
- [5] C. Darwin, “On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life,” *New York: D. Appleton*, 1859.
- [6] J. Felsenstein, “The number of evolutionary trees,” *Systematic Biology*, vol. 27, no. 1, p. 27, 1978.
- [7] L. Osborn, “Number of species identified on earth,” 2010. [Online]. Available: <http://www.currentresults.com/Environment-Facts/Plants-Animals/number-species.php>
- [8] J. Felsenstein, “Inferring phylogenies (2004) Sunderland,” *Massachusetts: Sinauer Associates*, vol. 90.
- [9] H. Bodlaender, M. Fellows, and T. Warnow, “Two strikes against perfect phylogeny,” *Automata, Languages and Programming*, pp. 273–283, 1992.
- [10] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis*. Cambridge university press Cambridge, UK:, 2002.
- [11] J. Felsenstein, “Evolutionary trees from DNA sequences: a maximum likelihood approach,” *Journal of molecular evolution*, vol. 17, no. 6, pp. 368–376, 1981.

- [12] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Addison-Wesley Professional, 2004.
- [13] A. Zomaya, *Parallel computing for bioinformatics and computational biology: models, enabling technologies, and case studies*. John Wiley and Sons, 2006.
- [14] J. Tuimala, "Using clustalx for multiple sequence alignment," 2004. [Online]. Available: http://www.csc.fi/english/research/sciences/bioscience/programs/clustalw/clustalx_pdf
- [15] D. Mount, *Bioinformatics: sequence and genome analysis*. CSHL press, 2004.
- [16] A. Stamatakis, T. Ludwig, and H. Meier, "RAxML-III: a fast program for maximum likelihood-based inference of large phylogenetic trees," *Bioinformatics*, vol. 21, no. 4, p. 456, 2005.
- [17] X. Feng, D. Buell, J. Rose, and P. Waddell, "Parallel algorithms for Bayesian phylogenetic inference," *Journal of Parallel and Distributed Computing*, vol. 63, no. 7-8, pp. 707–718, 2003.
- [18] M. Suchard and A. Rambaut, "Many-core algorithms for statistical phylogenetics," *Bioinformatics*, vol. 25, no. 11, p. 1370, 2009.
- [19] H. ConsortiumInternational, "Finishing the euchromatic sequence of the human genome," *Nature*, vol. 431, no. 7011, pp. 931–945, 2004.
- [20] S. Pond and S. Muse, "HyPhy: hypothesis testing using phylogenies," *Statistical Methods in Molecular Evolution*, pp. 125–181, 2005.
- [21] J. Seward, N. Nethercote, and J. Weidendorfer, *Valgrind 3.3-Advanced Debugging and Profiling for GNU/Linux applications*. Network Theory Ltd., 2008.
- [22] S. Guindon, F. Lethiec, P. Duroux, and O. Gascuel, "PHYML Online—a web server for fast maximum likelihood-based phylogenetic inference," *Nucleic acids research*, vol. 33, no. Web Server Issue, p. W557, 2005.
- [23] V. Heywood and J. McNeill, "Phenetic and phylogenetic classification," 1964.
- [24] E. Petterson, J. Lundeberg, and A. Ahmadian, "Generations of sequencing technologies," *Genomics*, vol. 93, no. 2, pp. 105–111, 2009.
- [25] Ø. Olsvik, J. Wahlberg, B. Petterson, M. Uhlen, T. Popovic, I. Wachsmuth, and P. Fields, "Use of automated sequencing of polymerase chain reaction-generated amplicons to identify three types of cholera toxin

- subunit B in *Vibrio cholerae* O1 strains.” *Journal of clinical microbiology*, vol. 31, no. 1, p. 22, 1993.
- [26] A. Smith, S. Datta, G. Smith, P. Campbell, R. Bentley, H. McKenzie, D. Bender, A. Harris, T. Goodwin, J. Parish *et al.*, *Oxford dictionary of biochemistry and molecular biology*. Oxford University Press London, 1997.
- [27] W. Gilbert, “Origin of life: The RNA world,” *Nature*, vol. 319, no. 6055, 1986.
- [28] T. Creighton, *Proteins: structures and molecular properties*. WH Freeman, 1993.
- [29] J. Pevsner, *Bioinformatics and functional genomics*. Blackwell Pub, 2009.
- [30] I. C. on Zoological Nomenclature, *International code of zoological nomenclature*. International Trust for Zoological Nomenclature, 1999.
- [31] R. Sedgewick, *Algorithms in c++, parts 1-4: fundamentals, data structure, sorting, searching*. Addison-Wesley Professional, 1998.
- [32] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, “High-throughput sequence alignment using Graphics Processing Units,” *BMC bioinformatics*, vol. 8, no. 1, p. 474, 2007.
- [33] P. Lio and N. Goldman, “Models of molecular evolution and phylogeny,” *Genome Research*, vol. 8, no. 12, p. 1233, 1998.
- [34] D. Culler, J. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann Pub, 1999.
- [35] G. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.
- [36] J. Neumann, “First Draft of a Report on the EDVAC,” *University of Pennsylvania*, 1945.
- [37] A. Silberschatz, P. Galvin, and G. Gagne, “Operating system concepts,” 2008.
- [38] W. Stallings, *Computer organization and architecture: designing for performance*. Prentice Hall, 2009.
- [39] A. Bhatele, L. Wesolowski, E. Bohm, E. Solomonik, and L. Kale, “Understanding Application Performance via Micro-benchmarks on Three Large Supercomputers: Intrepid, Ranger and Jaguar,” *International Journal of High Performance Computing Applications*, 2010.

- [40] H. Schmidt, K. Strimmer, M. Vingron, and A. Von Haeseler, "TREE-PUZZLE: maximum likelihood phylogenetic analysis using quartets and parallel computing," *Bioinformatics*, vol. 18, no. 3, p. 502, 2002.
- [41] B. Moret, D. Bader, and T. Warnow, "High-performance algorithm engineering for computational phylogenetics," *The Journal of Supercomputing*, vol. 22, no. 1, pp. 99–111, 2002.
- [42] X. Feng, K. Cameron, C. Sosa, and B. Smith, "Building the tree of life on terascale systems," in *Proceedings of the 21st International Parallel and Distributed Processing Symposium, Long Beach, CA*. Citeseer, 2007.
- [43] L. Vinh and A. Von Haeseler, "IQPNNI: moving fast through tree space and stopping in time," *Molecular biology and evolution*, vol. 21, no. 8, p. 1565, 2004.
- [44] H. Schmidt, E. Petzold, M. Vingron, and A. Von Haeseler, "Molecular phylogenetics: parallelized parameter estimation and quartet puzzling," *Journal of Parallel and Distributed Computing*, vol. 63, no. 7-8, pp. 719–727, 2003.
- [45] A. Stamatakis, "RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models," *Bioinformatics*, vol. 22, no. 21, p. 2688, 2006.
- [46] Z. Du, F. Lin, and U. Roshan, "Reconstruction of large phylogenetic trees: a parallel approach," *Computational Biology and Chemistry*, vol. 29, no. 4, pp. 273–280, 2005.
- [47] A. Stamatakis, T. Ludwig, H. Meier, and M. Wolf, "Accelerating parallel maximum likelihood-based phylogenetic tree calculations using subtree equality vectors," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2002, pp. 1–16.
- [48] Z. Yang, *Computational molecular evolution*. Oxford University Press, USA, 2006.
- [49] M. Goodrich and R. Tamassia, *Algorithm Design: Foundation, Analysis and Internet Examples*. Wiley India Pvt. Ltd., 2006.
- [50] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, 1969.
- [51] J. Felsenstein, "The newick tree format." [Online]. Available: <http://evolution.genetics.washington.edu/phylip/newicktree.html>
- [52] S. Russel and P. Norvig, "Artificial Intelligence: A Modern Approach," *Prentice Hall*, 2009.

- [53] Y. Bao, P. Bolotov, D. Dernovoy, B. Kiryutin, L. Zaslavsky, T. Tatusova, J. Ostell, and D. Lipman, "The influenza virus resource at the national center for biotechnology information," *Journal of Virology*, vol. 82, no. 2, p. 596, 2008.
- [54] C. Kuiken, B. Foley, T. Leitner, C. Apetrei, B. Hahn, I. Mizrachi, J. Mullins, A. Rambaut, S. Wolinsky, and B. Korber, "HIV Sequence Compendium," 2010. [Online]. Available: <http://www.hiv.lanl.gov/content/immunology/>
- [55] P. Lemey, A. Rambaut, and O. Pybus, "HIV evolutionary dynamics within and among hosts," *AIDS Rev*, vol. 8, no. 3, pp. 125–140, 2006.
- [56] E. Holmes, "The comparative genomics of viral emergence," *Proceedings of the National Academy of Sciences*, vol. 107, no. suppl 1, p. 1742, 2010.
- [57] B. Néron, H. Ménager, C. Maufrais, N. Joly, J. Maupetit, S. Letort, S. Carrere, P. Tuffery, and C. Letondal, "Mobylye: a new full web bioinformatics framework," *Bioinformatics*, vol. 25, no. 22, p. 3005, 2009.
- [58] I. Pasteur, "Mobylye web bioinformatics framework," 2009. [Online]. Available: <http://mobylye.pasteur.fr/>
- [59] A. Rambaut, "FigTree, a graphical viewer of phylogenetic trees," See <http://tree.bio.ed.ac.uk/software/figtree>, 2007.
- [60] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.